Curves: OpenGL Evaluators - Intro

- Evaluators are used to describe polynomials, splines, curved surfaces of any degree
- Based on Bézier curves
- If want to use another curve form, programmer must provide code to convert other form’s basis to Bézier basis
- OpenGL provides evaluators for 1D and 2D
- Also provides high-level interface for NURBS (via GLU)
  - Interface implemented in terms of evaluators
- \( Q(t) = T \cdot M_B \cdot G_B \)
  - Programmer supplies \( G_B \) and granularity of \( t \)
  - Evaluator plots the curve
- Evaluator can also generate color info, normals, and texture coordinates for points on the curve/surface
- To use evaluators:
  1. Define evaluator
  2. Enable evaluator
  3. Draw the curve
Curves: OpenGL Evaluators - 1D

- 1D evaluators use a single parameter
- Specify curves
- Steps in using:
  1. Defining:

        void glMap1{fd} (GLenum target, TYPE u1, TYPE u2, GLint stride, GLINT order, const TYPE *points);

- target values and semantics:

<table>
<thead>
<tr>
<th>Value</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>GL_MAP1_VERTEX_3</td>
<td>(x, y, z) vertex</td>
</tr>
<tr>
<td>GL_MAP1_VERTEX_4</td>
<td>(x, y, z, w) vertex</td>
</tr>
<tr>
<td>GL_MAP1_INDEX</td>
<td>color index</td>
</tr>
<tr>
<td>GL_MAP1_COLOR_4</td>
<td>R, G, B, A</td>
</tr>
<tr>
<td>GL_MAP1_NORMAL</td>
<td>[x y z]</td>
</tr>
<tr>
<td>GL_MAP1_TEXTURE_COORD_1</td>
<td>s tex coord</td>
</tr>
<tr>
<td>GL_MAP1_TEXTURE_COORD_2</td>
<td>s, t tex coords</td>
</tr>
<tr>
<td>GL_MAP1_TEXTURE_COORD_3</td>
<td>s, t, r tex coords</td>
</tr>
<tr>
<td>GL_MAP1_TEXTURECOORD_4</td>
<td>s, t, r, q tex coords</td>
</tr>
</tbody>
</table>

- points: Array holding values to be processed (CPs, colors, normals, etc.
- u1, u2: Initial and final values of curve parameter
- stride: Number of values separating successive data items in points - Can have several types of data intermingled in one data array
- order: Order of curve
  * Order is degree + 1
  * Cubic order is 4 (number of CPs)
- Note:
  (a) Each call to glMap1* creates an evaluator for the given target type
  (b) Color, tex coords, etc. for curve must be defined in terms of evaluators

2. Enabling
   - Evaluators enabled using glEnable with one of the target values used in glMap1*
   - Can only have 1 VERTEX and 1 TEXCOORD evaluator enabled at any time
Curves: OpenGL Evaluators - 1D (2)

3. Drawing

(a) Method 1: Specifying values of $u$ explicitly

```c
void glEvalCoord1\{fd\}(TYPE $u$);
void glEvalCoord1\{fd\}v(TYPE *$u$);
```

- Causes evaluation of all currently-enabled 1D maps at value $u$
- `glEvalCoord*` should be placed between `glBegin/glEnd` pair

(b) Method 2: Automatic generation of regular grid

- Done in 2 steps:
  i. Specify the parameters to be used to draw the curve
     ```c
     void glMapGrid1\{fd\}(GLint $n$, TYPE $u1$, TYPE $u2$);
     ```
     * Indicates curve is to be drawn from $u1$ to $u2$ in $n$ evenly-spaced intervals
  ii. Draw the curve
     ```c
     void glEvalMesh1(GLenum $mode$, GLint $p1$, GLint $p2$);
     ```
     * Mode: GL\_POINT, GL\_LINE
     * $p1$, $p2$: Delimiting values of $u$ over which curve is to be drawn
     \[0 \leq p1, p2 \leq n\]
     * `glEvalMesh1` applies the defined grid to all enabled evaluators
Curves: OpenGL Evaluators - 2D

- Extension of 1D evaluators
- Steps in using:
  1. Defining:

```c
void glMap2{fd} (GLenum target, TYPE u1, TYPE u2, GLint ustride,
            TYPE v1, TYPE v2, GLint vstride,
            GLint order, const TYPE *points);
```

- `target` values and semantics:

<table>
<thead>
<tr>
<th>Value</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>GL_MAP2_VERTEX_3</td>
<td>(x, y, z) vertex</td>
</tr>
<tr>
<td>GL_MAP2_VERTEX_4</td>
<td>(x, y, z, w) vertex</td>
</tr>
<tr>
<td>GL_MAP2_INDEX</td>
<td>color index</td>
</tr>
<tr>
<td>GL_MAP2_COLOR_4</td>
<td>R, G, B, A</td>
</tr>
<tr>
<td>GL_MAP2_NORMAL</td>
<td>[x \ y \ z]</td>
</tr>
<tr>
<td>GL_MAP2_TEXTURE_COORD_1</td>
<td>s tex coord</td>
</tr>
<tr>
<td>GL_MAP2_TEXTURE_COORD_2</td>
<td>s, t tex coords</td>
</tr>
<tr>
<td>GL_MAP2_TEXTURE_COORD_3</td>
<td>s, t, r tex coords</td>
</tr>
<tr>
<td>GL_MAP2_TEXTURE_COORD_4</td>
<td>s, t, r, q tex coords</td>
</tr>
</tbody>
</table>

- `points`: As in `Map1`
- `u1`, `u2`: As in `Map1`
  * `u` direction left to right
- `ustride`: Number of values separating successive data items in `u` dimension in `points`
- `uorder`: Order of curve in `u` direction
- `v1`, `v2`, `vorder`: `v` direction versions of `u1`, `u2`, `uorder`
  * `v` direction bottom to top
  * Order in `u` and `v` can be different

2. Enabling

- As in 1D, using MAP2 types
Curves: OpenGL Evaluators - 2D (2)

3. Drawing

(a) Specifying values of \( u, v \) explicitly

- Use
  
  ```c
  void glEvalCoord2(fd)(TYPE u, TYPE v);
  void glEvalCoord2(fd)v(TYPE *values);
  ```

- Analogous to 1D version

- When VERTEX evaluator enabled, normals automatically generated when GL_AUTO_NORMAL property in effect

  * Otherwise, programmer must supply normals using \( \text{Map2} \) evaluator

  * If no normal map is provided, current normal is used

(b) Automatic generation of regular grid

i. Specify the parameters to be used to draw the curve

  ```c
  void glMapGrid2(fd)(GLint nu, TYPE u1, TYPE u2, GLint nv, TYPE v1, TYPE v2);
  ```

- Analogous to \( \text{Map1} \) version

- Defines an \( nu \times nv \) grid

ii. Draw the curve

  ```c
  void glEvalMesh2(GLenum mode, GLint i1, GLint i2, GLint j1, GLint j2);
  ```

- Mode: GL_POINT, GL_LINE, GL_FILL

- \( i1, i2 \): start and end steps along \( u \)

- \( j1, j2 \): start and end steps along \( v \)
Curves: OpenGL Evaluators - Textures

• To use texture mapping, need a separate 2D Bézier patch

• Patch is planar:

\[
\begin{array}{c}
(0,0) \\
(1,0) \\
(0,1) \\
(1,1)
\end{array}
\]

• Patch is defined by 4 CPs
  – CPs defined in terms of \( x \) and \( y \) only
Curves: OpenGL NURBS Interface - Intro

• NURBS interface provide by the GLU
• Implemented in terms of evaluators
• Steps in using are similar to those used in tessellation
  – Create a NURBS rendering object
  – Register callback functions
  – Set/enable properties
  – Start the surface
  – Generate the curve
  – End the surface
  – Deallocate the rendering object
Curves: OpenGL NURBS Interface - Creating and Destroying

• To create a NURBS object:
  
  `void GLUnurbsObj* gluNewNurbsRenderer (void);`

  – If creation is unsuccessful, 0 is returned

• To destroy a NURBS object:
  
  `void gluDeleteNurbsRenderer (GLUnurbsObj *nobj);`
Curves: OpenGL NURBS Interface - Properties

- Major difference between tessellation and NURBS programming is that
  - In tessellation, callbacks are primary way in which characteristics of tessel-
    lators are controlled; setting properties is secondary and not required
  - In NURBS, properties are primary way in which characteristics of NURBS
    are controlled; callbacks are secondary and not required

- To set NURBS properties:
  
  ```c
  void gluNurbsProperty (GLUnurbsObj *obj, GLenum property,
                           GLfloat value);
  ```
  
  - `property` and values:

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>GLU_DISPLAY_MODE</td>
<td>GLU_FILL (default)</td>
</tr>
<tr>
<td></td>
<td>GLU_OUTLINE_POLYGON</td>
</tr>
<tr>
<td></td>
<td>GLU_OUTLINE_PATCH</td>
</tr>
<tr>
<td>GLU_NURBS_MODE</td>
<td>GLU_NURBS_RENDERER (default)</td>
</tr>
<tr>
<td></td>
<td>GLU_NURBS_TESSELLATOR</td>
</tr>
<tr>
<td>GLU_CULLING</td>
<td>GL_FALSE (default)</td>
</tr>
<tr>
<td></td>
<td>GLU_TRUE</td>
</tr>
<tr>
<td>GLU_SAMPLING_METHOD</td>
<td>GLU_PATH_LENGTH (default)</td>
</tr>
<tr>
<td></td>
<td>GLU_PARAMETRIC_ERROR</td>
</tr>
<tr>
<td></td>
<td>GLU_DOMAIN_DISTANCE</td>
</tr>
<tr>
<td></td>
<td>GLU_OBJECT_PATH_LENGTH</td>
</tr>
<tr>
<td></td>
<td>GLU_OBJECT_PARAMETER_ERROR</td>
</tr>
<tr>
<td>GLU_SAMPLING_TOLERANCE</td>
<td>float (default 50.0)</td>
</tr>
<tr>
<td>GLU_PARAMETRIC_TOLERANCE</td>
<td>float (default 0.5)</td>
</tr>
<tr>
<td>GLU_U_STEP</td>
<td>int (default 100)</td>
</tr>
<tr>
<td>GLU_V_STEP</td>
<td>(default 100)</td>
</tr>
<tr>
<td>GLU_AUTO_LOAD_MATRIX</td>
<td>GLU_TRUE</td>
</tr>
<tr>
<td></td>
<td>GLU_FALSE</td>
</tr>
</tbody>
</table>
Curves: OpenGL NURBS Interface - Properties (2)

* **DISPLAY_MODE:** Controls how surface displayed
  - FILL renders surface as polygons
  - OUTLINE_POLYGON renders as outlines of polygons created by NURBS tessellation
  - OUTLINE_PATCH renders only outlines of patches and trimming curves

* **NURBS_MODE:** Controls whether surface is drawn automatically
  - NURBS_RENDERER causes surface to be drawn
  - NURBS_TESSELLATOR returns results of tessellation to callback functions

* **CULLING:** Turns culling on and off

* **SAMPLING_METHOD:** Controls how \( u, v \) are sampled
  - **PATH_LENGTH:** Limits number of pixels per edge to be \( \leq \) value set by SAMPLING_TOLERANCE
  - **PARAMETRIC_ERROR:** Limits distance in pixels between tessellated polygon and actual surface to be \( \leq \) value set by PARAMETRIC_TOLERANCE
  - **DOMAIN_DISTANCE:** Specifies number of sampling points per unit length in \( u, v \) as given by U_STEP and V_STEP
  - **OBJECT_PATH_LENGTH:** As PATH_LENGTH, but distance specified in terms of object space, not pixels
  - **OBJECT_PARAMETER_ERROR:** As PARAMETRIC_ERROR, but distance specified in terms of object space, not pixels

* **GLU_SAMPLING_TOLERANCE:** See SAMPLING_METHOD
* **GLU_PARAMETRIC_TOLERANCE:** See SAMPLING_METHOD
* **GLU_U_STEP, GLU_V_STEP:** Control number of sampling points per unit length

* **GLU_AUTO_LOAD_MATRIX:** Specifies whether projection, model view, and view port mapping matrices are downloaded from the server (GL_TRUE) or will be supplied by *gluLoadSamplingMatrices* (See below)
Curves: OpenGL NURBS Interface - Properties (3)

- To manually control viewing aspects:
  
  ```
  gluLoadSamplingMatrices (GLUnurbsObj *obj,
      const GLfloat modelMatrix[16],
      const GLfloat projMatrix[16],
      const GLint viewport[4]);
  ```

- To determine current values of properties:
  
  ```
  void gluGetNurbsProperty (GLUnurbsObj *obj, GLenum property,
       GLfloat *value);
  ```
Curves: OpenGL NURBS Interface - Drawing Curves

• Drawing delimited by *begin/end* pair:
  
  void gluBeginCurve (GLUnurbsObj *obj);
  void gluEndCurve (GLUnurbsObj *obj);
  
  – These save and restore underlying evaluator state

• Drawing commands:

  void gluNurbsCurve (GLUnurbsObj *obj, GLint uknot_count,
            GLfloat *uknot, GLint u_stride,
            GLfloat *ctlArray, GLint worder,
            GLenum type);

  – *type*: Specifies what the data represents
    * Uses same values as 1D evaluators (e.g., GL_MAP1_VERTEX_3)
  – *worder*: Order of the curve
  – *ctlArray*: Array of CPs
  – *uknot_count*: How many knots in the curve
  – *u_stride*: Number of values between successive CPs

• By default, NURBS evaluator automatically generates curves and surfaces and draws them without programmer needing to specify callbacks
Curves: OpenGL NURBS Interface - Callbacks

• To register callbacks:

```c
void gluNurbsCallback (GLUnurbsObj *obj, GLenum type, void (*fn)() foo);
```

– Valid types and corresponding function prototypes:

<table>
<thead>
<tr>
<th>Type</th>
<th>Prototype</th>
</tr>
</thead>
<tbody>
<tr>
<td>GLU_NURBS_BEGIN</td>
<td>void begin (GLenum type);</td>
</tr>
<tr>
<td>GLU_NURBS_BEGIN_DATA</td>
<td>void begin (GLenum type, void *user_data);</td>
</tr>
<tr>
<td>GLU_NURBS_END</td>
<td>void begin (void);</td>
</tr>
<tr>
<td>GLU_NURBS_END_DATA</td>
<td>void begin (void *user_data);</td>
</tr>
<tr>
<td>GLU_NURBS_VERTEX</td>
<td>void begin (void *vertex_data);</td>
</tr>
<tr>
<td>GLU_NURBS_VERTEX_DATA</td>
<td>void begin (void *vertex_data, void *user_data);</td>
</tr>
<tr>
<td>GLU_NURBS_COLOR</td>
<td>void begin (GLfloat *color);</td>
</tr>
<tr>
<td>GLU_NURBS_COLOR_DATA</td>
<td>void begin (GLfloat *color, void *user_data);</td>
</tr>
<tr>
<td>GLU_NURBS_NORMAL</td>
<td>void begin (GLfloat *normals);</td>
</tr>
<tr>
<td>GLU_NURBS_NORMAL_DATA</td>
<td>void begin (GLfloat *normals, void *user_data);</td>
</tr>
<tr>
<td>GLU_NURBS_TEXTURE_COORD</td>
<td>void begin (void *tex_coords);</td>
</tr>
<tr>
<td>GLU_NURBS_TEXTURECOORD_DATA</td>
<td>void begin (void *tex_coords, void *user_data);</td>
</tr>
<tr>
<td>GLU_NURBS_ERROR</td>
<td>void begin (GLenum errno);</td>
</tr>
</tbody>
</table>

• `gluNurbsCallback` works the same way as `gluTessCallback`

• Drawing mode controlled by NURBS_MODE property

  – NURBS_RENDERER mode generates image automatically
    * The only callback that will be active (if registered) is ERROR
  – NURBS_TESSELLATOR mode inhibits automatic image generation
    * All registered callbacks are in effect
    * Programmer needs to include appropriate calls in callbacks in order to generate the image
    * This mode provided so programmer has access to values generated by NURBS tessellator
Curves: OpenGL NURBS Interface - Drawing Surfaces

• Drawing delimited by `begin/end` pair:
  1. For curves:
     ```
     void gluBeginCurve (GLUnurbsObj *obj);
     void gluEndCurve (GLUnurbsObj *obj);
     ```
  2. For surfaces:
     ```
     void gluBeginSurface (GLUnurbsObj *obj);
     void gluEndSurface (GLUnurbsObj *obj);
     ```
  3. These save and restore underlying evaluator state

• Drawing commands:
  1. For curves:
     ```
     void gluNurbsCurve (GLUnurbsObj *obj, GLint uknot_count,
                         GLfloat *uknot, GLint u_stride,
                         GLfloat *ctlArray, GLint uorder,
                         GLenum type);
     ```
     – `type`: Specifies what the data represents
       * Uses same values as 1D evaluators (e.g., GL_MAP1_VERTEX_3)
     – `uorder`: Order of the curve
     – `*ctlArray`: Array of CPs
     – `*uknot_count`: How many knots in the curve
     – `u_stride`: Number of values between successive CPs
  2. For surfaces:
     ```
     void gluNurbsSurface (GLUnurbsObj *obj,
                           GLint uknot_count, GLfloat *uknot,
                           GLint vknout_count, GLfloat *vknout,
                           GLint u_stride, GLint v_stride,
                           GLfloat *ctlArray,
                           GLint uorder, GLint vorder,
                           GLenum type);
     ```
     – Parameters same as for curves in both u and v directions

• By default, NURBS tessellator automatically generates curves and surfaces and
draws them without programmer needing to specify callbacks
Curves: OpenGL NURBS Interface - Trimming

- Trimming somewhat comparable to contours in tessellation
  - Like contours, trimming allows creation of holes in NURBS surface
  - Unlike contours, they are not required

- Trimming calls appear between begin/end delimiters of NURBS surface

- Trimming itself delimited by begin/end pair:
  
  void gluBeginTrim (GLUnurbsObj *, obj);
  void gluEndTrim (GLUnurbsObj *, obj);

- Trimming specified in terms of closed curves
  - Curves must be closed manually
  - Trimming curve may be constructed piecewise from many individually-defined curve segments
    
    * Programmer must insure that end point of one segment is start point of next

- Orientation of curve determines how trim is effected
  - As follow curve from start to end, points to the left are retained, points to the right are excluded (trimmed)
  - If trimming curves are nested, programmer must insure that inconsistencies do not exist
  
  - Trimming curves cannot intersect

![Diagram of a surface with a hole illustrating trimming](image-url)
Curves: OpenGL NURBS Interface - Trimming (2)

- Types of trimming curves

1. NURBS
   - Created with `gluNurbsCurve`
   - Must be contained within $0 \leq u, v \leq 1$
   - Type must be GLU_MAP1_TRIM_2 or GLU_MAP1_TRIM_3
     * TRIM_2 is usual mode
     * TRIM_3 includes homogeneous coordinate

2. Piecewise linear curve
   - Creates a curve approximated by straight line segments
   - Specified using
     ```
     void gluPwlCurve (GLUnurbsObj *obj, GLint count, GLfloat *array, 
                      GLint stride, GLenum type);
     ```
     * `array`: Array of points on the curve
     * `count`: Number of points on the curve
     * `type`: GLU_MAP1_TRIM_2 or GLU_MAP1_TRIM_3
     * `stride`: Number of values between successive points in `array` (dependent on TRIM_2 or TRIM_3)