Special Topics in OpenGL
Rasterization
What is Rasterization?

• Is a process by which a primitive is converted to a 2D image
  ➢ Determine which squares of an integer grid in window coordinates are occupied by the primitive
  ➢ Assign a color and a depth value to each square

• A grid square along with its assigned color and depth is called a fragment

• The results of the process are passed to the next stage of per-fragment operations
Point Rasterization

- Point is rasterized as a single fragment truncating its \((x_w, y_w)\) coordinates to integers

- For wide points, fragment centers are at

\[
\left(\left\lfloor x_w \right\rfloor + \frac{1}{2}, \left\lfloor y_w \right\rfloor + \frac{1}{2}\right) \quad \text{(Odd)}
\]

\[
\left(\left\lceil x_w + \frac{1}{2} \right\rceil, \left\lceil y_w + \frac{1}{2} \right\rceil\right) \quad \text{(Even)}
\]

- The data associated with each rasterized fragment is the same as that of the vertex
Line Segment Rasterization

- Diamond-exit rule to determine the fragments produced by rasterization

- Specify the data associate with each rasterized fragment

\[ f = \frac{(1 - t) f_a / w_a + tf_b / w_b}{(1 - t) / w_a + t / w_b} \]

Where \[ t = \frac{(p - p_a) \cdot (p_b - p_a)}{|p_b - p_a|^2} \]
Polygon Rasterization

- **Point sampling**
  - Rasterized fragment centers lie inside the projected polygon
  - If two or more polygons share the same fragment, it is rasterized by one of them

- **Specify the data associated with each rasterized fragment**

\[
f = \frac{af_a / w_a + bf_b / w_b + cf_c / w_c}{a / w_a + b / w_b + c / w_c}
\]

\[
p = ap_a + bp_b + cp_c \text{ defines any point in a triangle with barycentric coordinates (a, b, and c)}
\]
Antialiasing

- Lines (nearly horizontal or vertical) appear zagged

- Reducing this zaggedness is called antialiasing
  - Calculates a coverage value for each fragment based on the fraction of the pixel square on the screen that it would occur
  - Multiplies the fragment’s alpha by its coverage
  - Use the resulting alpha to blend the fragment with the corresponding pixel already in the frame buffer

- Antialiasing points or lines or polygons
  - Pass GL_POINT_SMOOTH or GL_LINE_SMOOTH or GL_POLYGON_SMOOTH to glEnable()
  - Enable blending

Example: Aliased and antialiased lines
Framebuffer
What is Framebuffer

- Each fragment has coordinate data which correspond to a pixel, as well as color and depth values.

- Buffers (storages) to hold the various kinds of information of pixels.

- OpenGL implementation supports the following buffers:
  - Color buffer
  - Depth buffer
  - Stencil buffer
  - Accumulation buffer

  The buffers are used to perform special tasks before pixels are finally written to the viewable color buffer.

- A collection of these buffers is called framebuffer.
Color Buffers

• Color buffers are the ones to which you draw
  ➢ They contain RGBA data

• Stereoscopic viewing needs left and right color buffers for the left and right stereo images

• Double-buffered systems have front and back color buffers

• Non-displayable auxiliary color buffers can be used

• Minimum requirement is a front-left color buffer
Other Buffers

• Depth buffer (z-buffer):
  ➢ Stores a depth value for each pixel
  ➢ Depth is usually measured in terms of distance to the eye
  ➢ Used for a hidden-surface removal

• Stencil buffer:
  ➢ Stores the information to restrict drawing to certain portions of the screen

• Accumulation buffer:
  ➢ Holds RGBA color data for accumulating a series of images into a final, composite image
  ➢ When accumulation is finished, the result is copied back into the color buffer for viewing
  ➢ Used for scene antialiasing, motion blur, simulating depth of field, and calculating soft shadows
Clearing Buffers

- Clearing the screen (or any of the buffers) is expensive
  - Hardware can clear more than one buffer at once

- First, specify the current clearing values for each buffer
  
  ```c
  void glClearColor(GLclampf red, GLclampf green, GLclampf blue, GLclampf alpha);
  void glClearDepth(GLclampf depth);
  void glClearStencil(GLuint s);
  void glClearAccum(GLclampf red, GLclampf green, GLclampf blue, GLclampf alpha);
  ```

- Then issue a single clear command
  
  ```c
  void glClear(GLbitfield mask);
  
  mask is the bitwise logical OR of some combination of
  GL_COLOR_BUFFER_BIT, GL_DEPTH_BUFFER_BIT, GL_STENCIL_BUFFER_BIT, and GL_ACCUM_BUFFER_BIT
  ```
Color Buffers for Writing and Reading

• **void glDrawBuffer(GLenum mode);**
  - Selects the color buffers enabled for writing or clearing
  - `mode` can be GL_FRONT, GL_BACK, GL_LEFT, GL_RIGHT, GL_FRONT_LEFT, etc.
  - Default mode is GL_FRONT for single-buffered contexts and GL_BACK for double-buffered contexts

• **void glReadBuffer(GLenum mode);**
  - Selects the color buffer enabled as the source for reading pixels
Masking Buffers

- Sets the masks used to control writing into the indicated buffers

- void `glColorMask(GLboolean red, GLboolean green, GLboolean blue, GLboolean alpha);
  - The `red`, `green`, `blue` and `alpha` values control whether corresponding component is written

- void `glDepthMask(GLboolean flag);
  - `flag` is `GL_TRUE` for writing in depth buffer

- void `glStencilMask(GLuint mask);
  - `mask = 1` for writing the bit
Testing and Operating on Fragments

• After fragments are generated, several processing stages occur determining how and whether a given fragment is drawn as pixel into the framebuffer

• Set of tests:
  ➢ Scissor test
  ➢ Alpha test
  ➢ Depth test
  ➢ Stencil test
  ➢ Blending
  ➢ Dithering
  ➢ Logical operation
Scissor Test

- void `glScissor(Glint x, Glint y, GLsizei width, GLsizei height);`
  - Sets the location and size of the scissor rectangle or box
  - By default, the rectangle matches the window
  - Drawing occurs only inside the rectangle: pixels lying inside the rectangle pass the scissor test
  - Needs enabling
    `glEnable(GL_SCISSOR_TEST);`
Alpha Test

- `void glAlphaFunc(GLenum func, GLclampf ref);`
  - Sets the reference value and comparison function for the alpha test
  - In RGBA mode, a fragment is accepted or rejected by the alpha test on its alpha value
  - By default, `ref` is zero, and `func` is GL_ALWAYS
  - `func` can be GL_ALWAYS, GL_NEVER, GL_LESS, GL_EQUAL, GL_LEQUAL, GL_GEQUAL, GL_GREATER or GL_NOTEQUAL
  - Needs enabling
    
    ```
    glEnable(GL_ALPHA_TEST);
    ```
Depth Test

- `glDepthFunc(GLenum func);`
  - Sets the comparison function for the depth test
  - An incoming fragment passes the depth test if its z value has specified relation to the value already stored in the depth buffer
  - By default, `func` is GL_LESS
    Pixels with larger depth-buffer values are overwritten by pixels with smaller values

  - `func` can be GL_ALWAYS, GL_EQUAL, GL_GREATER, etc.

  - Needs enabling
    `glEnable(GL_DEPTH_TEST);`
Stencil Test

- The stencil test takes place only if there is a stencil buffer
  - It compares a reference value with the value stored at a pixel in the buffer
  - Depending on the test result, the value in the stencil buffer is modified

- `void glStencilFunc(GLenum func, GLint ref, GLuint mask);`
  - Sets the comparison `func`, reference `ref` and `mask` for the test
    - Comparison applies to those bits for which bits of the mask are 1
  - `func` can be `GL_ALWAYS`, `GL_LESS`, etc.
  - Needs enabling: `glEnable(GL_STENCIL);`

- `glStencilOp(GLenum fail, GLenum zfail, GLenum zpass);`
  - Specifies how the data in the stencil buffer is modified when a fragment passes or fails the stencil test
  - `fail`, `zfail`, `zpass` can be `GL_KEEP`, `GL_ZERO`, `GL_REPLACE`, `GL_INCR`, `GL_DECR`, `GL_INVERT`

  \[
  fail = \text{failed stencil test}; \quad zfail = \text{failed z test}; \quad zpass = \text{passed z test}
  \]
Other Operations

• **Blending**
  ➢ Combines the incoming fragment’s R, G, B and A values with those of the pixel already stored at the location

• **Dithering**
  ➢ Dither the values of red, green and blue on neighboring pixels for the perception of a wide range of colors
  Needs enabling with GL_DITHER

• **Logical Operations**
  ➢ Are applied between the incoming fragment’s color and the color stored at the corresponding location in the framebuffer
  ➢ The result replaces the value in the framebuffer for that fragment

```c
void glLogicOp(GLenum opcode);
```

`opcode` can be GL_CLEAR, GL_COPY, GL_AND, etc

Needs enabling with GL_COLOR_LOGIC
Blending
What Blending?

• Combining colors from a source (incoming fragment) and destination (the corresponding pixel) to achieve such effects as making objects appear translucent

• The source and destination factors are RGBA quadruplets:
  - (Sr, Sg, Sb, Sa) and (Dr, Dg, Db, Da)

• Blended RGBA values are
  - (RsSr + RdDr, GsSg + GdDg, BsSb+BdDb, AsSa+AdDa)

  Where (Rs,Gs,Bs,As) and (Rd,Gd,Bd,Ad) are the RGBA values of source and destination
How to Specify?

- `void glBlendFunc(GLenum sfactor, GLenum dfactor);`
  - Controls how color values in the fragment being processed (the source) are combined with those already stored in the framebuffer (the destination)
  - `sfactor (dfactor)` indicates how to compute a source (destination) blending factor
    - `GL_ONE`: (1,1,1,1)
    - `GL_SRC_ALPHA`: (As,As,As,As)
    - `GL_ONE_MINUS_SRC_ALPHA`: (1,1,1,1)-(As,As,As,As)

- Needs enabling
  - `glEnable(GL_BEND);`
3D Blending with the Depth Buffer

- For 3D objects, the appearance depends on whether you draw the polygons from back to front or from front to back
  - drawing order

- Consider the effect of the depth buffer in determining the order
  - If an opaque object hides other objects, eliminate the more distant objects

- If the translucent object is closer, blend it with the opaque object

- **Example: Sphere inside a Cube**
Animation
Pictures That Move

- Animation is an important part of computer graphics
- Seeing all sides of a mechanical part designed
- Learning to fly an airplane using a simulation
- Viewing molecular dynamics
- Viewing vector data
Motion = Redraw + Swap

- OpenGL provides double buffering (two color buffers)
  - One is displayed while the other is being drawn
  - When drawing of a frame is complete, the two buffers are swapped
  - Like a movie projector with only two frames in a loop

- void `glutSwapBuffers(void);`
  - Swap the viewable and drawable buffers
  - Waits until one frame is completely drawn and other is completely displayed
  - For a system with display refresh rate of 60 times per second, the fastest frame rate can be 60 frames per second

- void `glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);`
  - Set double buffered window mode
Example: Solar System

```c
void display (void) {
    glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glColor3f (1.0, 1.0, 1.0);
    glPushMatrix ();
    glRotatef (year, 0.0, 1.0, 0.0);
    glutSolidSphere (1.0, 80, 64);
    glTranslatef (2.0, 0.0, 0.0);
    glRotatef (day, 0.0, 1.0, 0.0);
    glutSolidSphere (0.2, 80, 64);
    glPopMatrix ();
    glutSwapBuffers ();
}
```

```c
glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGB);
```

- Sun is rotating about its own axis; planet is orbiting around the sun as well as rotating about its own axis
- The graphics remain idle between the frames
Hidden-Surface Removal
In a scene composed of 3D objects, some of them might obscure all or parts of others.

The obscuring relationship changes with viewpoint and needs to be properly maintained.

Hidden-surface removal is elimination of parts of solid objects that are obscured by others.

Otherwise, the objects are drawn in the order the drawing commands appear in the code.

Hidden-surface removal increases performance.
Use of Depth Buffer

- Use of depth buffer (z-buffer) to achieve hidden surface removal

- Graphical calculations convert each surface (before drawing) to a set of corresponding pixels on the window and also compute depth value for each pixel

- A comparison is done with the depth value already stored at that pixel to accept the pixel only if it has a smaller depth

- Color and depth information of the incoming pixel with greater depth is discarded
How to Specify?

• In void `glDepthFunc(Glenum func);`
  Defualt value of `func` is used: `GL_LESS` used
  `glEnable(GL_DEPTH_TEST);`

• `glutInitDisplayMode (GLUT_RGB | GLUT_DEPTH);`

• Before drawing, each time you need to clear the depth buffer and draw objects in any order
  `glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);`

  `glClear()` clears both color and depth buffers

• **Planet hides behind the sun in solar system example**
Drawing Pixel Data
Geometric Versus Pixel Data

- Rendering of geometric data (arrays of vertices)
  - points, lines, polygons

- Rendering of pixel data (arrays of pixels)
  - Bitmaps
    - Characters in fonts
    - Array of 0s and 1s ($1 = \text{the pixel is affected}$)
    - Serves as drawing mask for overlying another image

  - Image data
    - A photograph that is scanned or an image calculated by some program in memory by pixels or an image first drawn and then read back pixel by pixel
    - Several pieces of data per pixel (R,G,B,A values)
    - Simply overwrites in the framebuffer
Current Raster Position

- void `glRasterPos{234}{sifd}(TYPE x, TYPE y, TYPE z, TYPE w);`
  - Sets the current raster position where the next bitmap (or image) is to be drawn
  - The raster position coordinates are subject to the modelview and projection transformations in the same way as the vertex coordinates

- To specify the raster position directly in the screen coordinates, only 2D version of transformations need to be specified
Drawing Bitmaps

- void glBitmap(GLsizei width, GLsizei height, GLfloat x_{bo}, GLfloat y_{bo}, GLfloat x_{bi}, GLfloat y_{bi}, const Glubyte *bitmap);
  - Draws the bitmap specified by bitmap
  - Width and height define size of the bitmap
  - Subscript bo means the origin of the bitmap
  - Subscript bi means increment to the current raster position after the bitmap is rasterized
Manipulating Images

- **void glReadPixels(GLint x, GLint y, GLsizei width, GLsizei height, GLenum format, GLenum type, GLvoid *pixels)**
  - Reads pixel data from the *specified* framebuffer rectangle and stores data in the array pointed by pixels
  - *format* can be GL_RGBA, GL_RED, GL_ALPHA, GL_DEPTH_COMPONENT
  - *type* can be s, u, i, f, etc.

- **void glDrawPixels(GLsizei width, GLsizei height, GLenum format, GLenum type, GLvoid *pixels)**
  - Draws a rectangle of pixel data with dimensions *width* and *height*
  - Pixel rectangle has its lower-left corner at the current raster position

- **void glCopyPixels(GLint x, GLint y, GLsizei width, GLsizei height, GLenum buffer)**
  - Copies pixel data from the specified framebuffer rectangle
  - Buffer can be GL_COLOR, GL_DEPTH, GL_STENCIL
Example: Drawing Image

- Make a checkerboard image

- Define raster position

- Draw an pixel rectangle of the image
Texture Mapping
What is Texture Mapping?

- Gluing an image (such as scanned photograph) to a polygon
  - Bricks on wall
  - Ground in flight simulation
  - Vegetation

- Textures are rectangular arrays of data (colors, luminance)
  - Individual values are called texels

- Textures can be manipulated with transformations
  - Repeat textures in different directions to cover the surface
  - Apply textures in different shapes and sizes
Steps in Texturing

- Create a texture object and specify a texture for the object
- Indicate how the texture is to be applied to each pixel
- Enable texture mapping
- Draw the scene by supplying both texture and geometric coordinates
Sample Example

- Checkboard texture is generated
  ```
  makeCheckImage()
  ```

- All texture mapping initialization occurs in `init()`
  ```
  glGenTextures(1, &texName);
  glBindTexture(GL_TEXTURE_2D, texName);
  ```

- Single, full resolution texture map is specified
  ```
  glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, checkImageWidth, checkImageHeight, 0, GL_RGBA, GL_UNSIGNED_BYTE, checkImage);
  ```

- Specify how the texture to be wrapped and how the colors are to be filtered if there is not an exact match between texels and pixels
  ```
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
  ```
More on Example

• In `display(void)`, texture is turned on
  
  ```
  glEnable(GL_TEXTURE_2D);
  ```

• Drawing mode is set so as to draw the textured polygons using the colors from the texture map
  
  ```
  glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE);
  ```

• Two polygons are drawn by specifying texture coordinates along with vertex coordinates
  
  ```
  glTexCoord2f(0.0,0.0);  glVertex3f(-2.0,-1.0,0.0);
  ```

• Texture is finally turned off
  
  ```
  glDisable(GL_TEXTURE_2D);
  ```
3D Textures

- 3D textures are used in scientific visualization
  - e.g. volume rendering

- Defining a 3D texture:
  - `glTexImage3D(GL_TEXTURE_3D, 0, GL_RGB, iWidth, iHeight, iDepth, 0, GL_RGB, GL_UNSIGNED_BYTE, image);`

- Replace all or some of the texels of a 3D texture

- Using compressed texture images

- Mipmaps: Multiple levels of detail

- Filtering

- Texture objects

- Texturing functions
Display Lists
What and Why Display List?

• In many cases, you may need to execute the same set of OpenGL commands multiple times

• Drawing a tricycle:
  ➢ Two wheels on the back are the same size but are offset from each other. The front wheel is larger and in a different location
  ➢ An efficient way to render the wheels on the tricycle is store the geometry for one wheel in a list and then execute the list three times by setting the appropriate modelview matrix each time before executing the list

• Several other examples: Solar system, molecular dynamics
What and Why Display List?

- A display list is a group of OpenGL commands that have been stored for later execution.

- You can define the geometry and/or state changes once and execute them multiple times by providing a number that uniquely specifies the display list.

- Display lists improve performance by caching commands which are reused many times.
Naming and Creating a Display List

- Each display list is identified as a unique, system-generated integer index or ID

- GLuint glGenLists (GLuint range);
  - Allocates range number of contiguous, previously unallocated display list indices
  - The integer returned is the index that marks the beginning of a contiguous block of empty display list indices

    If returned integer is n, then indices n, n+1, ……, n + range -1 are available

    listIndex = glGenLists(1);

    Generates one new display list ID and store it in variable listIndex

    glIsList (GLuint list) to check whether a specific index is in use. It returns GL_TRUE if the list is already used

    glDeleteLists (GLuint list, GLsizei range) to delete range display lists starting at the index specified by list
Naming and Creating a Display List

- **void glNewList**(GLuint list, GLenum mode);
  - Specifies the start of a display list.
  - The argument *list* is a nonzero positive integer that uniquely identifies the display list.
  - The possible values for *mode* are GL_COMPILE and COMPILE_EXECUTE.

- **void glEndList**(void);
  - Marks the end of a display list.

  ```c
  glNewList(listIndex, GL_COMPILE);
  ........
  glEndList();
  ```
Executing a Display List

• **Void glVertexList(GLint list);**
  - Executes the display list specified by `list` which is the index for the display list
  - Commands in the display list are executed, just as if they were issued

• You can execute the same display list many times

• You can mix display lists and immediate-mode graphics
Example

- The display list contains OpenGL commands to draw a triangle

- The display list is executed multiple times

- A rectangle is drawn in immediate mode.
Hierarchical Display Lists

- A hierarchical display list executes another display list in it by calling `glCallList()` between a `glNewList()` and `glEndList()` pair

  ➢ A display list to render a tricycle:

```c
glNewList(listIndex, GL_COMPLIE);
    glCallList(handlebars);
    glCallList(frame);
    glTranslatef(1.0,0.0,0.0);
    glCallList(wheel);
    glTranslatef(3.0,0.0,0.0);
    glCallList(wheel);
    glTranslatef(3.0,0.0,0.0);
    glScalef(1.5,1.5,1.5);
    glCallList(wheel);
    glEndList();
```
Multiple Display Lists

• **void glListBase(GLuint base);**
  - Specifies the offset that’s added to the display-list indices in glCallLists() to obtain the final display-list indices

• **void glCallLists(GLsizei n, GLenum type, const GLvoid *lists);**
  - Executes $n$ display lists
  - *lists* is a pointer that points to an array of offsets
  - Nesting level of display lists is at least 64
Managing State Variables

- A display list can contain calls that change the value of OpenGL state variables

- The changes persist after execution of the display list is completed

- Use `glPushMatrix()` to save a group of state variables and `glPopMatrix()` to restore the values later

- Use `glPushMatrix()` and `glPopMatrix()` to save and restore the current matrix
Selection
Interactive Applications

• Allows user to select a region of the scene or pick an object drawn on the screen

• Selection mode
  ➢ First draw scene, then use selection mode, and redraw the scene
  ➢ Screen remains frozen while OpenGL is in selection mode
  ➢ On exiting from selection mode, OpenGL returns a list of primitives that intersect the viewing volume
  ➢ Each primitive within the viewing volume causes a selection hit
Basic Steps

- Specify the array to be used for the returned hit records with \texttt{glSelectBuffer()}

- Enter selection mode by specifying \texttt{GL_SELECT} with \texttt{glRenderMode()}

- Initialize the name stack using \texttt{glInitNames()}

- Define viewing volume to be used for selection

- Exit selection mode and process the hit records
Commands

• **void glSelectBuffer(GLsizei size, GLuint *buffer);**
  - Specifies the array to be used for the returned selection data
  - `buffer` is a pointer to the array of the given `size`

• **void glRenderMode(GLenum mode);**
  - Controls whether the application is in rendering, selection, or feedback mode
  - `mode` is GL_RENDER, GL_SELECT or GL_FEEDBACK
  - `mode` remains unchanged until `glRenderMode()` is called again with different argument
Creating the Name Stack

- **void glInitNames(void);**
  - Clears the name stack so that it’s empty

- **void glPushName(GLuint mode);**
  - Pushes name onto the name stack
  - The stack contain at least 64 names

- **void glPopName(void);**
  - Pops one name off the top of the name stack

- **void glLoadName(GLuint name);**
  - Replaces the value on the top of the name stack with name
Hit Record

• A primitive that intersects the viewing volume causes a selection hit

• OpenGL writes a hit record into the selection array if there is a hit

• Each hit record consists of the following items
  - Number of names on the name stack when the hit is occurred
  - Minimum and maximum window coordinate depth (z) values of all selected primitives
  - Contents of the name stack at the time of the hit
Picking

- Use selection mode to determine if the object are picked

- Picking is triggered by an input device (mouse click)

- Use a special picking matrix in conjunction with the projection matrix

```c
void glPickMatrix(GLdouble x, GLdouble y, GLdouble width, GLdouble height, GLint viewport[4]);
```

- Creates a projection matrix that restricts drawing to a small region of the viewport and multiplies that matrix onto current matrix stack
- `x, y` define the center of picking region (or cursor location)
- `width and height` define the size of the picking region
- `viewport[]` indicates the current viewport boundaries.
Evaluators
What is Evaluators?

- Provide way to describe curves and surfaces by using few parameters or control points

- Use a polynomial mapping to produce vertex, normal, and texture coordinates, and colors

- Precision and storage efficient
One-Dimensional Evaluators

A vector-valued function (called Bezier curve) of one variable is

\[ C(u) = [X(u), Y(u), Z(u)] = \sum_{i=0}^{n} B_i^n(u)P_i \]

Where \( P_i \) represent a set of \( n \) control points (3D) for vertices, colors or normals, and

\[ B_i^n(u) = \binom{n}{i} u^i(1-u)^{n-i} \]

is \( i^{th} \) Bernstein polynomial of degree \( n \)

Domain for variable \( u \) is \([0.0, 1.0]\).

But if it is \([u_1, u_2]\), the function at \( u \) is evaluated as

\[ C \left( \frac{u - u_1}{u_2 - u_1} \right) \]

Example: distance traveled by a body as a function of time.
Defining

- **void glMap1{fd}**(GLenum *target*, TYPE *u*<sub>1</sub>, TYPE *u*<sub>2</sub>, GLint *stride*, GLInt *order*, const TYPE *points*);
  - Defines 1D evaluator
  - *target*: what control point represent and how many values need to be supplied in points
    - GL_MAP1_VERTEX_3, GL_MAP1_COLOR_4, GL_MAP1_NORMAL
  - *u*<sub>1</sub> and *u*<sub>2</sub>: range for variable *u*
  - *stride*: an offset value between the beginning of one control point and the beginning of the next
  - *order*: degree + 1

```c
glMap1f(GL_MAP1_VERTEX_3, 0.0, 1.0, 3, 4, &cntrlpoints[0][0]);
glEnable(GL_MAP1_VERTEX_3);
```
Evaluating

- \texttt{void glEvalCoord1\{fd\}(TYPE u);}:
  - Causes evaluation of the enabled 1D maps
  - \( u \) is the domain coordinate
  - Call does not affect the current values for color and normal vectors
  - Call appears between \texttt{glBegin()} and \texttt{glEnd()} pair

- More than one evaluator can be evaluated at a time
  - Define and enable both \texttt{GL\_MAP1\_VERTEX\_3} and \texttt{GL\_MAP1\_COLOR\_4}
  - so that a single call to \texttt{glEvalCoord1()} generate both position and color along the curve
Defining Evenly Spaced $u$ Values

- Use a 1D grid of $u$ values for evaluation of function

- void MapGrid1(fd)(GLint $n$, TYPE $u_1$, TYPE $u_2$);
  - Define a grid that goes from $u_1$ and $u_2$ in $n$ steps, which are evenly spaced

- void glEvalMesh1(GLenum mode, GLint $p_1$, GLint $p_2$);
  - Applies currently defined map grid to all enabled evaluators
  - Mode: GL_POINT or GL_LINE
  - $p_1$ and $p_2$ defines the range of steps
Two-Dimensional Evaluators

A vector-valued function (called Bezier surface) of two variables \((u \text{ and } v)\) is

\[
C(u,v) = [X(u,v), Y(u,v), Z(u,v)] = \sum_{i=0}^{n} \sum_{j=0}^{m} B_i^n(u)B_j^m(v)P_{ij}
\]

Where \(P_{ij}\) represents a set of \(m*n\) control points (3D), and \(B's\) are Bernstein polynomials

Procedure to use 2D evaluators:

1. Define evaluator with \texttt{glMap2()} 
2. Enable them by passing appropriate value to \texttt{glEnable()} 
3. Invoke them either by \texttt{glEvalCoord2()} between \texttt{glBegin()} and \texttt{glEnd()} pair or
   - By specifying and applying a mesh with \texttt{glMapGrid2()} and \texttt{glEvalMesh2()}
2D Evaluators Command

- Void `glMap2{fd}(GLenum target, TYPE u_1, TYPE u_2, GLint ustride, GLint uorder, TYPE v_1, TYPE v_2, GLint vstride, GLint vorder, const TYPE *points);`

- Void `glEvalCoord2{fd}(TYPE u, TYPE v);`

- Void `glMapGrid2{fd}(GLint nu, TYPE u_1, TYPE u_2, GLint nv, TYPE v_1, TYPE v_2);`
  void `glEvalMesh2(GLenum mode, GLint i_1, GLint i_2, GLint j_1, GLint j_2);`
  `mode` can be GL_POINT, GL_LINE, GL_FILL

- Normal to the surface can be computed with `glEnable(GL_AUTO_NORMAL)`. 

CSC 7443: Scientific Information Visualization

BB Karki, LSU
Tessellators, Quadrics, NURBs
Polygon Tessellation

• Process of subdividing non-simple polygons (such as concave polygons, polygons with holes, polygons with intersecting edges) into simple convex polygons

• Steps in polygon tessellation:
  ➢ Create a new tessellation object with **gluNewTess()**
  ➢ Use **gluTessCallback()** to register callback functions to perform operations during the tessellation
  ➢ Specify tessellation properties by calling **gluTessProperty()**
  ➢ Create and render tessellated polygons by the contours
  ➢ Delete tessellation object with **gluDeleteTess()**
Quadrics

- Rendering spheres, cylinders, and disks:
  - Quadric surfaces are defined by
    \[ a_1x^2 + a_2y^2 + a_3z^2 + a_4xy + a_5yz + a_6xz + a_7x + a_8y + a_9z + a_{10} = 0 \]

- Steps in using quadrics object
  - Use `gluNewQuadric()` to create a quadrics object
  - Specifying rendering attributes with `gluQuadricOrientation()`, `gluQuadricDrawstyle()`, `gluQuadricNormals()`
  - Invoke the rendering routines for different quadric objects: `gluSphere()`, `gluCylinder()`, `gluDisk()`
  - Delete the quadric object with `gluDeleteQuadric()`
NURBS Interface

- GLU provides a NURBS (Non-Uniform-Rational-B-Spline) interfaces

- Steps to draw NURBS curves or surfaces
  - Use `gluNewNurbsRender()` to create a NURBS object
  - Start your curve or surface by calling `gluBeginCurve()` or `gluBeginSurface()`
  - Generate and render curve or surface with call to `gluNurbsCurve()` or `gluNurbsSurface()`
  - Call `gluNurbsProperty()` to choose rendering values such as number of polygons used
  - Call `gluNurbsCallback()` for different functions
  - Use lighting with `glEnable(GL_AUTO_NORMAL)`
  - Complete drawing with `gluEndCurve()` or `gluEndSurface()`
Summary
OpenGL Rendering Pipeline

Vertex data → Evaluators → Per-Vertex Operations & Primitive Assembly → Rasterization → Per-Fragment Operations → Framebuffer

Display List

Pixel data → Pixel Operations
Programmable Pipeline

- Graphics processors (GPUs) are programmable
  - Functionality of some of major units in the graphics pipeline can be altered by user programs which are executed in GPU

- Vertex shader
  - User programs to manipulate vertex properties

- Fragment shader
  - User programs to alter the processing of fragments

- Shading languages
  - The OpenGL Shading Language based on C
  - C for graphics: Cg
Programmable Pipeline: A Simple View

- All data (vertex, interpolated values and fragments) pass through a non-programmable part of the hardware
  - GPU registers store and transfer the data.
What Have We Covered?

- OpenGL Basics
  - OpenGL and related libraries, window management
- Drawing
  - Geometric primitive objects, vertex arrays, normal vectors, polygonal models of surfaces
- Viewing
  - Modelview transformations, projection transformations, viewport transformation, clipping, matrix stacks
- Color
  - Color perception, color functions, shading
- Lighting
  - Light sources, lighting models, material properties
- Special topics
  - Rasterization, framebuffer, animation, hidden-surface removal, blending, drawing pixel data, texture mapping, display lists, evaluators, selection
  - GPU programming