Introduction to OpenGL

CMSC 435

Assignments

- Assignment 1 done!
- Assignment 2 finalized
  - Bezier Patch Terrain

Introducing OpenGL

- Recall the rendering pipeline:
  - Transform geometry (object ⮜ world, world ⮜ eye)
  - Calculate surface lighting
  - Apply perspective projection (eye ⮜ screen)
  - Clip to the view frustum
  - Perform visible-surface processing
- Implementing all this is a lot of work
- *OpenGL* provides a standard implementation
  - So why study the basics?

OpenGL Design Goals

- SGI's design goals for OpenGL
  - High-performance (hardware-accelerated) graphics API
  - Some hardware independence
  - Natural, terse API with some built-in extensibility
- OpenGL has become a standard because
  - It doesn't try to do too much
    - Only renders the image, doesn't manage windows, etc.
    - No high-level animation, modeling, sound (!), etc.
  - It does enough
    - Useful rendering effects + high performance
  - It was promoted by SGI (& Microsoft, half-heartedly), is now promoted/supported by NVIDIA, ATI, etc.
  - It doesn't change every year (like DirectX, it's main competitor)

OpenGL: Conventions

- Functions in OpenGL start with **`gl`**
  - Most functions just **`gl`** (e.g., **`glColor()`**)
  - Functions starting with **`glu`** are utility functions (e.g., **`gluLookAt()`**)
    - Note that GLU functions can always be composed entirely from core GL functions
  - Functions starting with **`glut`** are from the GLUT (OpenGL Utility Toolkit) library, built on top of OpenGL and WGL (Windows) or X (Linux) for window management, mouse and keyboard events, etc.
    - Created and distributed as an entirely different library

OpenGL: Conventions

- Function names indicate argument type and number
  - Functions ending with **`f`** take floats
  - Functions ending with **`i`** take ints
  - Functions ending with **`b`** take bytes
  - Functions ending with **`ub`** take unsigned bytes
  - Functions that end with **`v`** take an array.
- Examples
  - **`glColor3f()`** takes 3 floats
  - **`glColor4fv()`** takes an array of 4 floats

OpenGL: Simple Use

- Open a window and attach OpenGL to it
- Set projection parameters (e.g., field of view)
- Setup lighting, if any
- Main rendering loop
  - Set camera pose with **`gluLookAt()`**
    - Camera position specified in world coordinates
  - Render polygons of model
    - Use modeling matrices to transform vertices in world coordinates

OpenGL: Simple Use

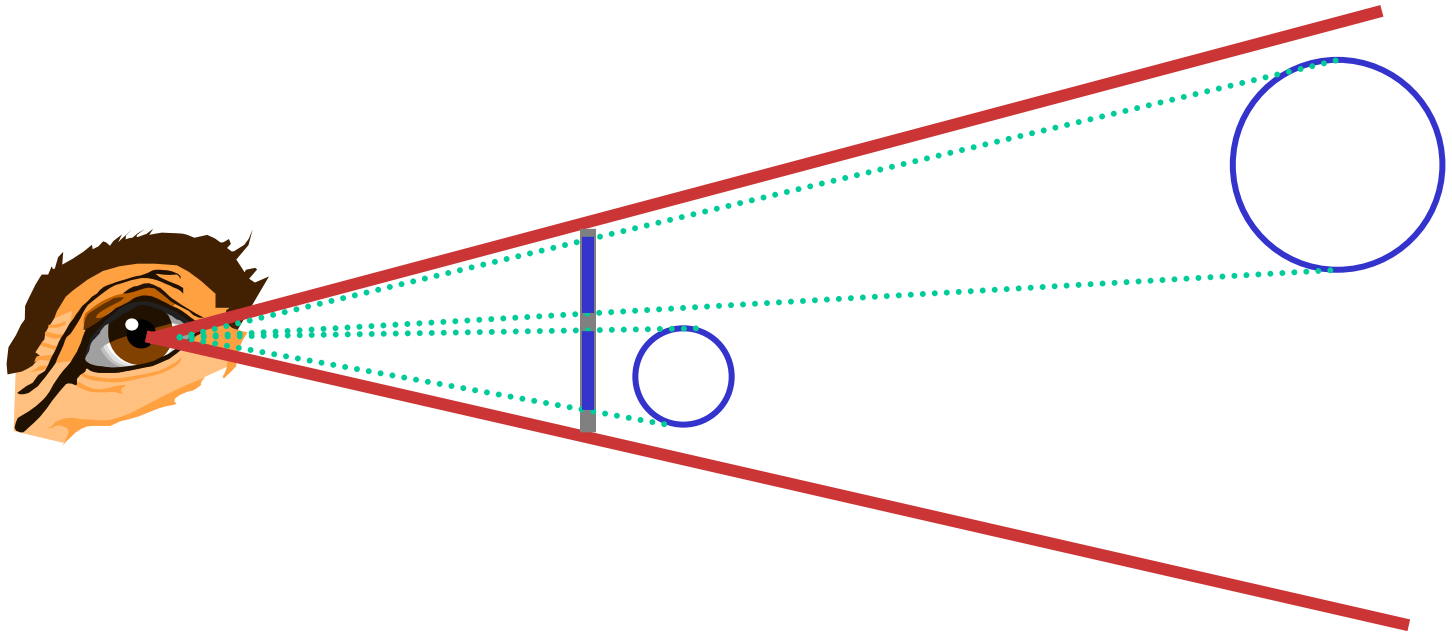
- *Open a window and attach OpenGL to it*
  - FLTK/GLUT
- Set projection parameters (e.g., field of view)
- Setup lighting, if any
- Main rendering loop
  - Set camera pose with **`gluLookAt()`**
    - Camera position specified in world coordinates
  - Render polygons of model
    - Use modeling matrices to transform vertices in world coordinates

OpenGL: Simple Use

- Open a window and attach OpenGL to it
- *Set projection parameters (e.g., field of view)*
- Setup lighting, if any
- Main rendering loop
  - Set camera pose with **`gluLookAt()`**
    - Camera position specified in world coordinates
  - Render polygons of model
    - Use modeling matrices to transform vertices in world coordinates

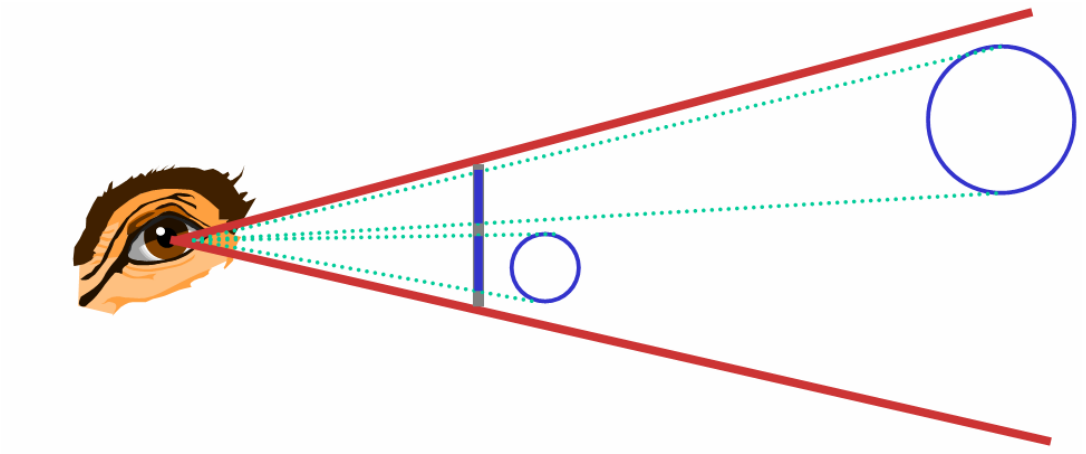
Projection Transform

- Projection transform
  - Apply perspective foreshortening
    - Distant = small: the *pinhole camera* model
  - View coordinates ⮜ screen coordinates



OpenGL: Perspective Projection

- Typically, we use a *perspective projection*
  - Distant objects appear smaller than near objects
  - Vanishing point at center of screen
  - Defined by a *view frustum* (draw it)
- Other projections: *orthographic*, *isometric*



OpenGL: Perspective Projection

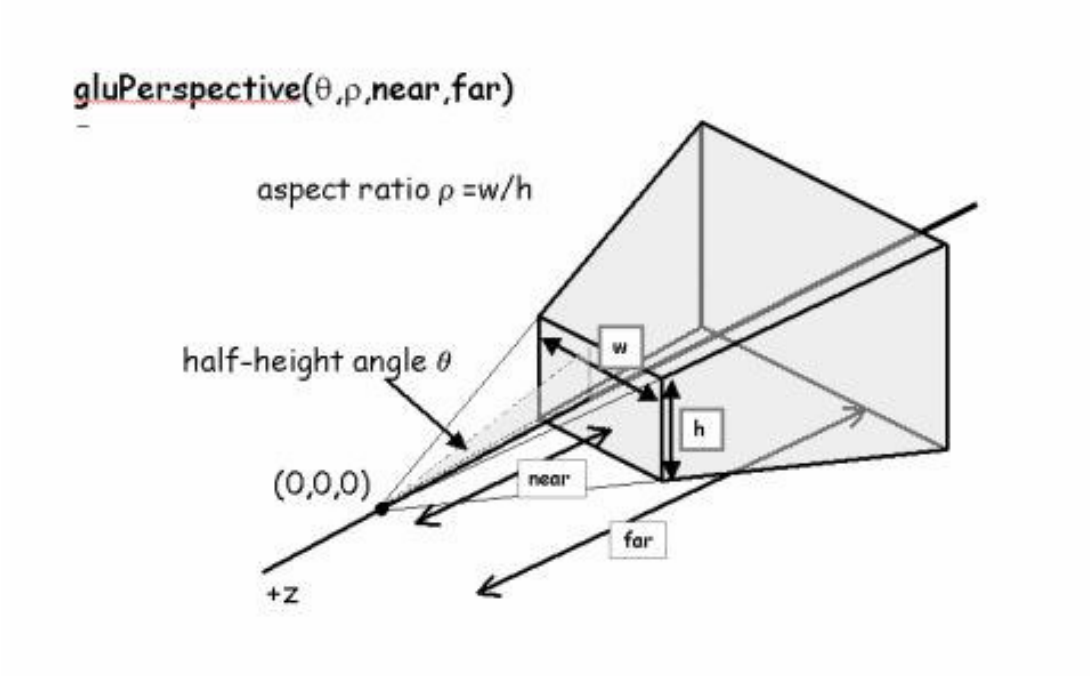
- In OpenGL:
  - Projections implemented by *projection matrix*
  - **`gluPerspective()`** creates a perspective projection matrix:

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluPerspective(vfov, aspect, near, far);
```

More on these in a bit

- Parameters to **`gluPerspective()`**:
  - **`vfov`**: vertical field of view
  - **`aspect`**: window width/height
  - **`near`**, **`far`**: distance to near & far clipping planes

OpenGL:Perspective Projection



- Parameters to **`gluPerspective()`**:
  - **`vfov`**: vertical field of view
  - **`aspect`**: window width/height
  - **`near`**, **`far`**: distance to near & far clipping planes

OpenGL: Simple Use

- Open a window and attach OpenGL to it
- Set projection parameters (e.g., field of view)
- *Setup lighting, if any*
- Main rendering loop
  - Set camera pose with **`gluLookAt()`**
    - Camera position specified in world coordinates
  - Render polygons of model
    - Use modeling matrices to transform vertices in world coordinates

OpenGL: Lighting

- Simplest option: change the *current color* between polygons or vertices
  - glColor() sets the current color
- Or OpenGL provides a simple lighting model:
  - Set parameters for light(s)
    - Intensity, position, direction & falloff (if applicable)
  - Set *material* parameters to describe how light reflects from the surface
- Won't go into details now; check the red book if interested

OpenGL: Simple Use

- Open a window and attach OpenGL to it
- Set projection parameters (e.g., field of view)
- Setup lighting, if any
- *Main rendering loop*
  - Set camera pose with **`gluLookAt()`**
    - Camera position specified in world coordinates
  - Render polygons of model
    - Use modeling matrices to transform vertices in world coordinates

OpenGL: Specifying Viewpoint

- `glMatrixMode(GL_MODELVIEW);`
- `glLoadIdentity();`

} *More on these in a bit*

- `gluLookAt(eyeX, eyeY, eyeZ,`
  `lookX, lookY, lookZ,`
  `upX, upY, upZ);`
  - **`eye[XYZ]:`** camera position in world coordinates
  - **`look[XYZ]:`** a point centered in camera's view
  - **`up[XYZ]:`** a vector defining the camera's vertical
- Creates a matrix that transforms points in world coordinates to *camera coordinates*
  - Camera at origin
  - Looking down -Z axis
  - Up vector aligned with Y axis (actually Y-Z plane)

OpenGL: Specifying Geometry

- Simple case first: object vertices already in world coordinates
- Geometry in OpenGL consists of a list of vertices in between calls to **`glBegin()`** and **`glEnd()`**
  - A simple example: telling GL to render a triangle

```
glBegin(GL_POLYGON);
    glVertex3f(x1, y1, z1);
    glVertex3f(x2, y2, z2);
    glVertex3f(x3, y3, z3);
glEnd();
```

  - Usage: **`glBegin(`***geomtype***`)`** where geomtype is:
    - Points, lines, polygons, triangles, quadrilaterals, etc...

OpenGL: More Examples

- Example: GL supports quadrilaterals:

```
glBegin(GL_QUADS);
   glVertex3f(-1, 1, 0);
   glVertex3f(-1, -1, 0);
   glVertex3f(1, -1, 0);
   glVertex3f(1, 1, 0);
glEnd();
```

  - This type of operation is called *immediate-mode rendering*; each command happens immediately

- *Why do you suppose OpenGL uses a series of* `glVertex()` *calls instead of one polygon function that takes all its vertices as arguments?*

OpenGL: Front/Back Rendering

- Each polygon has two sides, front and back
- OpenGL can render the two differently
- The ordering of vertices in the list determines which is the front side:
  - When looking at the *front* side, the vertices go *counterclockwise*
    - This is basically the right-hand rule
    - Note that this still holds after perspective projection

OpenGL: Drawing Triangles

- You can draw multiple triangles between **`glBegin(GL_TRIANGLES)`** and **`glEnd()`**:

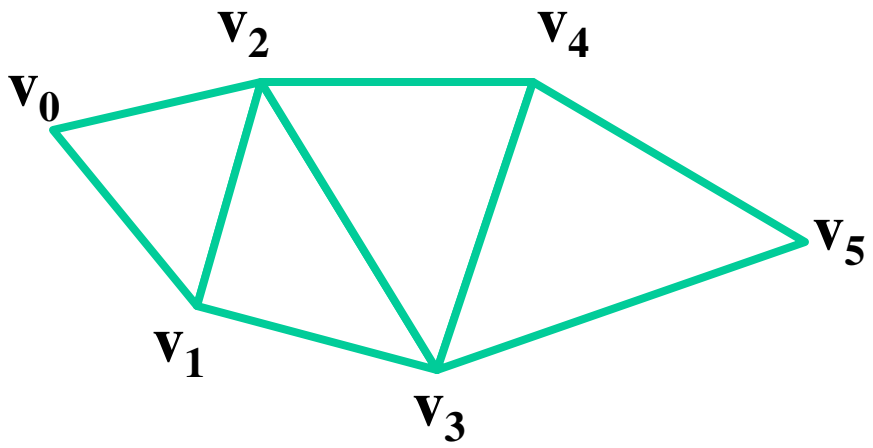
```
float v1[3], v2[3], v3[3], v4[3];
...
glBegin(GL_TRIANGLES);
glVertex3fv(v1); glVertex3fv(v2);
  glVertex3fv(v3);
glVertex3fv(v1); glVertex3fv(v3);
  glVertex3fv(v4);
glEnd();
```

- Each set of 3 vertices forms a triangle
  - *What do the triangles drawn above look like?*
  - *How much redundant computation is happening?*

OpenGL: Triangle Strips

- An OpenGL *triangle strip* primitive reduces this redundancy by sharing vertices:

```
glBegin(GL_TRIANGLE_STRIP);
glVertex3fv(v0);
glVertex3fv(v1);
glVertex3fv(v2);
glVertex3fv(v3);
glVertex3fv(v4);
glVertex3fv(v5);
glEnd();
```



  - triangle 0 is v0, v1, v2
  - triangle 1 is v2, v1, v3 (*why not v1, v2, v3?*)
  - triangle 2 is v2, v3, v4
  - triangle 3 is v4, v3, v5 (again, **not** v3, v4, v5)

OpenGL: Modeling Transforms

- OpenGL provides several commands for performing modeling transforms:
  - **`glTranslate{fd}(x, y, z)`**
    - Creates a matrix **T** that transforms an object by translating (moving) it by the specified *x*, *y*, and *z* values
  - **`glRotate{fd}(angle, x, y, z)`**
    - Creates a matrix **R** that transforms an object by rotating it counterclockwise *angle* degrees about the vector {*x*, *y*, *z*}
  - **`glScale{fd}(x, y, z)`**
    - Creates a matrix **S** that scales an object by the specified factors in the *x*, *y*, and *z* directions

OpenGL: Matrix Manipulation

- Each of these postmultiplies the *current matrix*
  - E.g., if current matrix is **C**, then **C=CS**
  - The current matrix is either the **`modelview`** matrix or the **`projection`** matrix (also a texture matrix, won't discuss)
  - Set these with **`glMatrixMode()`**, e.g.:

```
glMatrixMode(GL_MODELVIEW);
glMatrixMode(GL_PROJECTION);
```

  - **WARNING: common mistake ahead!**
    - Be sure that you are in **`GL_MODELVIEW`** mode before making modeling or viewing calls!
    - Ugly mistake because it can appear to work, at least for a while…

OpenGL: Matrix Manipulation

- More matrix manipulation calls
  - To replace the current matrix with an identity matrix:
    `glLoadIdentity()`
  - Postmultiply the current matrix with an arbitrary matrix:
    `glMultMatrix{fd}(float/double m[16])`
  - Copy the current matrix and push it onto a stack:
    `glPushMatrix()`
  - Discard the current matrix and replace it with whatever's on top of the stack:
    `glPopMatrix()`
  - Note that there are matrix stacks for both **`modelview`** and **`projection`** modes

OpenGL – Hello World

```
/*
 * hello.c
 * This is a simple, introductory OpenGL program.
 */
#include <GL/glut.h>

void display(void)
{
/* clear all pixels  */
   glClear (GL_COLOR_BUFFER_BIT);

/* draw white polygon (rectangle) with corners at
 * (0.25, 0.25, 0.0) and (0.75, 0.75, 0.0)
 */
   glColor3f (1.0, 1.0, 1.0);
   glBegin(GL_POLYGON);
      glVertex3f (0.25, 0.25, 0.0);
      glVertex3f (0.75, 0.25, 0.0);
      glVertex3f (0.75, 0.75, 0.0);
      glVertex3f (0.25, 0.75, 0.0);
   glEnd();
```

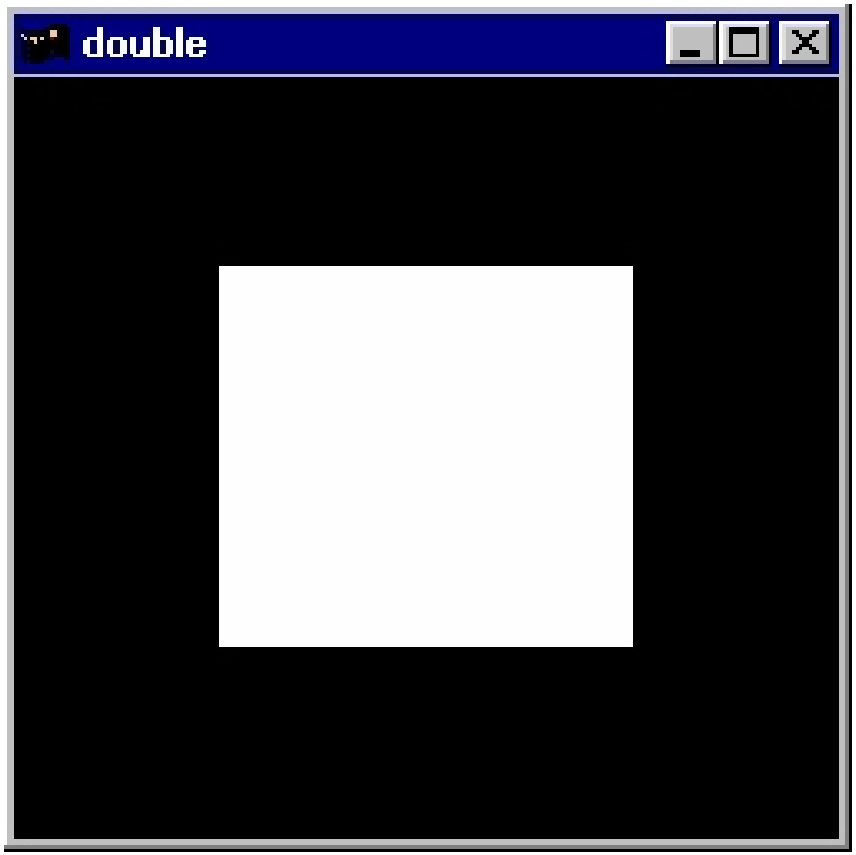
OpenGL – Hello World

```
/*
 * start processing buffered OpenGL routines
 */
   glFlush ();
}

void init (void)
{
/* select clearing color  */
   glClearColor (0.0, 0.0, 0.0, 0.0);

/* initialize viewing values  */
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   glOrtho(0.0, 1.0, 0.0, 1.0, -1.0, 1.0);
}
```

OpenGL – Hello World

```
/*
 * Declare initial window size, position, and display mode

 * (single buffer and RGBA).  Open window with "hello"
 * in its title bar.  Call initialization routines.
 * Register callback function to display graphics.
 * Enter main loop and process events.
 */
int main(int argc, char** argv)
{
   glutInit(&argc, argv);
   glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
   glutInitWindowSize (250, 250);
   glutInitWindowPosition (100, 100);
   glutCreateWindow ("hello");
   init ();
   glutDisplayFunc(display);
   glutMainLoop();
   return 0;   /* ANSI C requires main to return int. */
}
```

OpenGL Hello World



OpenGL Rotating Square

```
/*
 *  double.c
 *  This is a simple double buffered program.
 *  Pressing the left mouse button rotates the rectangle.
 *  Pressing the middle mouse button stops the rotation.
 */
#include <GL/glut.h>
#include <stdlib.h>

static GLfloat spin = 0.0;

void display(void)
{
   glClear(GL_COLOR_BUFFER_BIT);
   glPushMatrix();
   glRotatef(spin, 0.0, 0.0, 1.0);
   glColor3f(1.0, 1.0, 1.0);
   glRectf(-25.0, -25.0, 25.0, 25.0);
   glPopMatrix();

   glutSwapBuffers();
}
```

OpenGL Rotating Square

```
void spinDisplay(void)
{
   spin = spin + 2.0;
   if (spin > 360.0)
      spin = spin - 360.0;
   glutPostRedisplay();
}

void init(void)
{
   glClearColor (0.0, 0.0, 0.0, 0.0);
   glShadeModel (GL_FLAT);
}

void reshape(int w, int h)
{
   glViewport (0, 0, (GLsizei) w, (GLsizei) h);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   glOrtho(-50.0, 50.0, -50.0, 50.0, -1.0, 1.0);
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
}
```

OpenGL Rotating Square

```
void mouse(int button, int state, int x, int y)
{
   switch (button) {
      case GLUT_LEFT_BUTTON:
         if (state == GLUT_DOWN)
            glutIdleFunc(spinDisplay);
         break;
      case GLUT_MIDDLE_BUTTON:
      case GLUT_RIGHT_BUTTON:
         if (state == GLUT_DOWN)
            glutIdleFunc(NULL);
         break;
      default:
         break;
   }
}
```

OpenGL Rotation Square

```
/*
 *  Request double buffer display mode.
 *  Register mouse input callback functions
 */
int main(int argc, char** argv)
{
   glutInit(&argc, argv);
   glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGB);
   glutInitWindowSize (250, 250);
   glutInitWindowPosition (100, 100);
   glutCreateWindow (argv[0]);
   init ();
   glutDisplayFunc(display);
   glutReshapeFunc(reshape);
   glutMouseFunc(mouse);
   glutMainLoop();
   return 0;   /* ANSI C requires main to return int. */
}
```

OpenGL Rotating Square Demo

Trivia

- **Utah Teapot**
- The teapot was made by Melitta in 1974 and originally belonged to Martin Newell and his wife, Sandra - who purchased it from ZCMI, (a department store in Salt Lake City).