# CS418 Computer Graphics

Fall 2019, **Fall 2020**

**Omar Usman Khan** omar.khan@nu.edu.pk

Department of Computer Science National University of Computer & Emerging Sciences, Peshawar

December 20, 2020



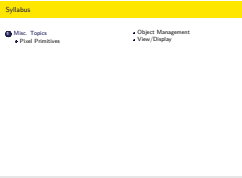



# Syllabus

1. Misc. Topics
   - Pixel Primitives
- Object Management
- View/Display





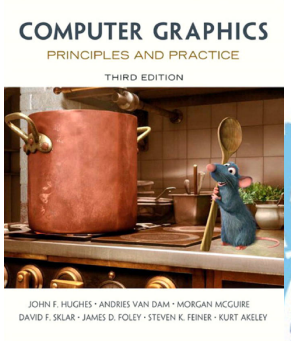
1. Misc. Topics
   - Pixel Primitives
   - Object Management
   - View/Display



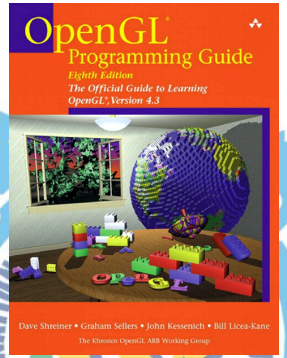



# Books


**D. Hearn et. al., Computer Graphics with OpenGL, 3rd edition**




J. F. Hughes et. al., Computer Graphics: Principles and Practice, third edition.

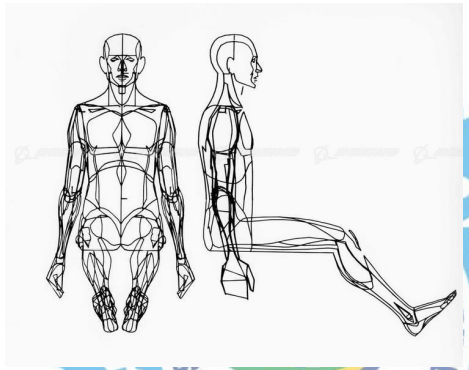



D. Shreiner et. al., OpenGL Programming Guide, Addison-Wesley, eighth edition.





# Computer Graphics

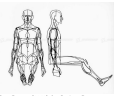


Coined by William Fetter (1928-2002) at Boeing (1960) for Cockpit Designs

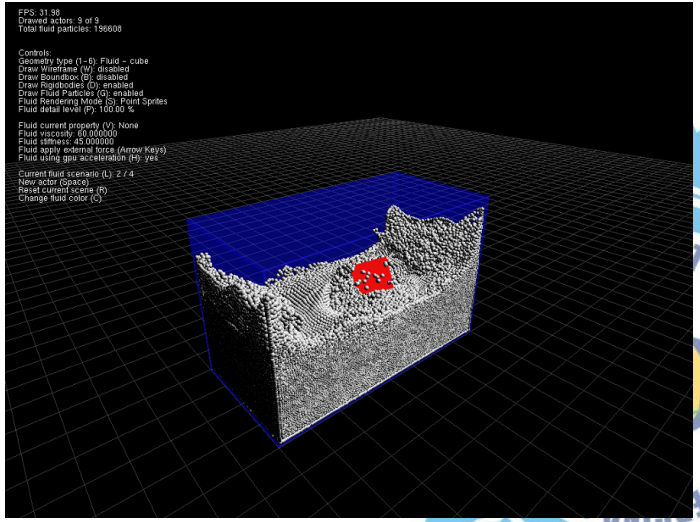




Computer Graphics


Coined by William Fetter (1928-2002) at Boeing (1960) for Cockpit Designs

# Applications

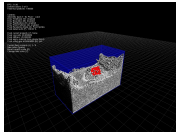



Scientific Visualizations (Simulations)





Applications



Scientific Visualizations (Simulations)

# Applications (cont.)

Arts & Design (Zaheer Mukhtar)





Applications (cont.)

Arts & Design (Zaheer Mukhtar)

# Applications (cont.)

Computer Games





Applications (cont.)

Computer Games

# Applications (cont.)

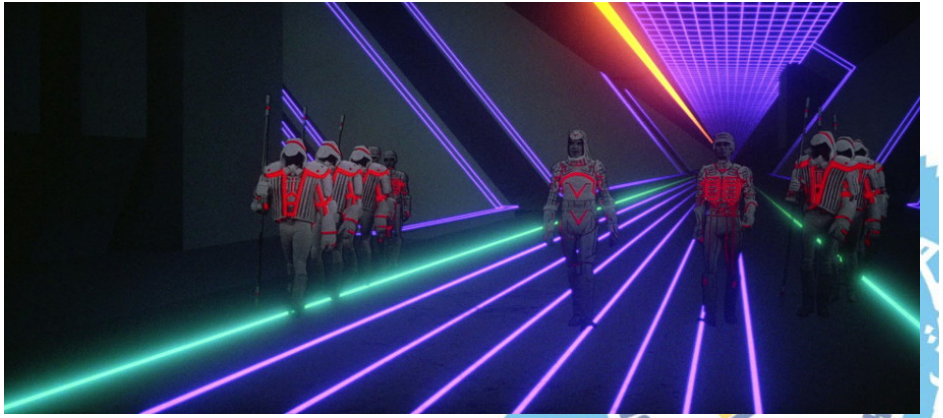
Architecture (Buildings) & Layout Design





Applications (cont.)
Architecture (Buildings) & Layout Design

# Applications (cont.)



Movie Industry (Tron, 1980)





Applications (cont.)

Movie Industry (Tron, 1980)

# Computer Graphics Field

- Science and Art of Communicating Visually

## Interdisciplinary

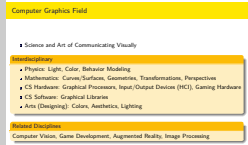
- Physics: Light, Color, Behavior Modeling
- Mathematics: Curves/Surfaces, Geometries, Transformations, Perspectives
- CS Hardware: Graphical Processors, Input/Output Devices (HCI), Gaming Hardware
- CS Software: Graphical Libraries
- Arts (Designing): Colors, Aesthetics, Lighting

## Related Disciplines

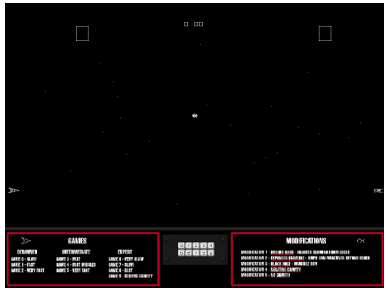
Computer Vision, Game Development, Augmented Reality, Image Processing



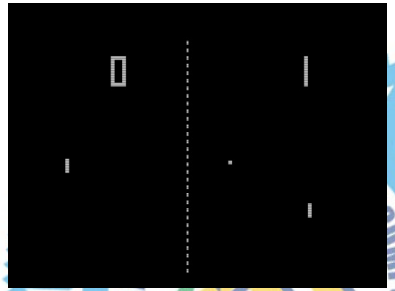




# Historical Firsts




Spacewars at MIT (1962)



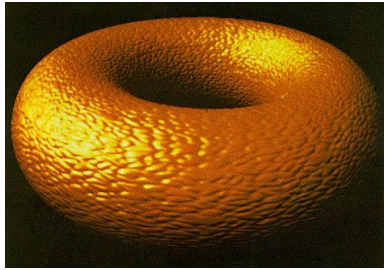
Pong for Atari (1973)





Historical Firsts
Spacewars at MIT (1962)
Pong for Atari (1973)

# Historical Firsts (cont.)



First Wrinkled/Bumped Image
(Blinn 1978)

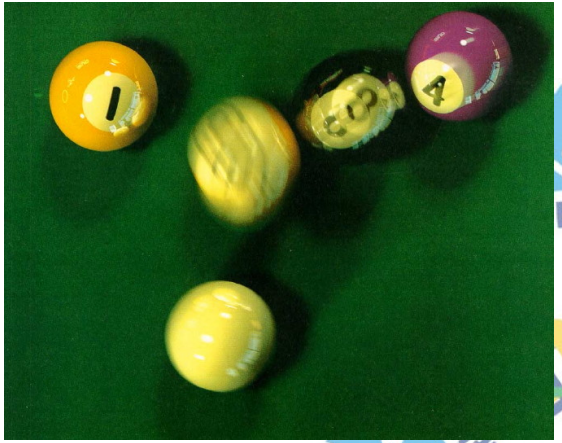
First Textured Image (Catmull 1974)





Historical Firsts (cont.)
First Wrinkled/Bumped Image (Blinn 1978)
First Textured Image (Catmull 1974)

# Historical Firsts (cont.)

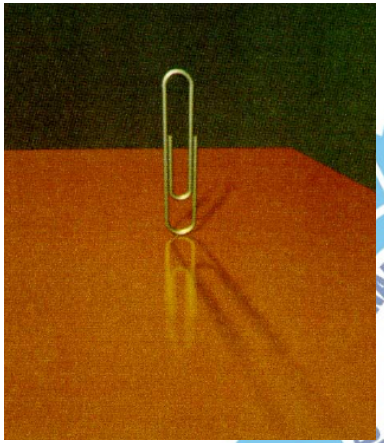






Historical Firsts (cont.)

# Historical Firsts (cont.)



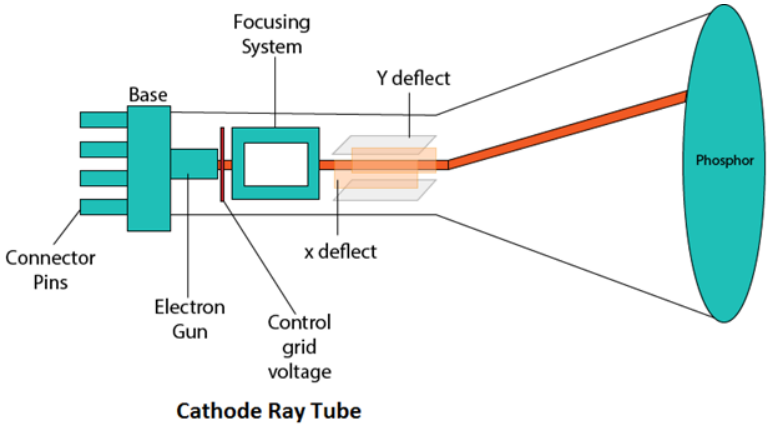
First raytraced images (Cook, 1984); motion blur, depth of field, translucency, reflections





Historical Firsts (cont.)
First raytraced images (Cook, 1984); motion blur, depth of field, translucency, reflections

# Display Devices

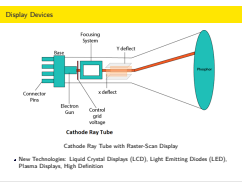



Cathode Ray Tube with Raster-Scan Display

- New Technologies: Liquid Crystal Displays (LCD), Light Emitting Diodes (LED), Plasma Displays, High Definition

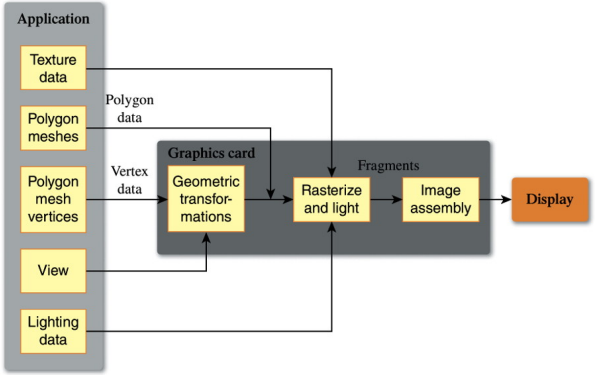






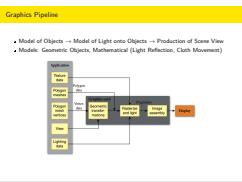
# Graphics Pipeline

- Model of Objects → Model of Light onto Objects → Production of Scene View
- Models: Geometric Objects, Mathematical (Light Reflection, Cloth Movement)







# Definitions

| | |
|---|---|
| Pixel | Smallest graphical object that can be drawn on a screen |
| Resolution | number of rows and columns of pixels |
| DPI | Dots per inch, packing capability of pixels in 1 inch space by the screen. |
| Pixel Position | Position of a pixel (row, col) with respect to screen coordinates |
| Clipping Area | A rectangular view on the screen where an application can draw objects |

## Rendering

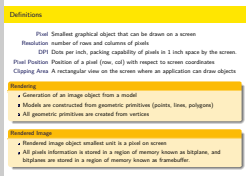
- Generation of an image object from a model
- Models are constructed from geometric primitives (points, lines, polygons)
- All geometric primitives are created from vertices

## Rendered Image

- Rendered image object smallest unit is a pixel on screen
- All pixels information is stored in a region of memory known as bitplane, and bitplanes are stored in a region of memory known as framebuffer.







# Screen Coordinate System

- 2D regular Cartesian grid
- (width $\times$ height) giving resolution
- Pixels at grid intersections
- origin (0,0) at upper left corner (platform convention) with reversed quadrants

## World View

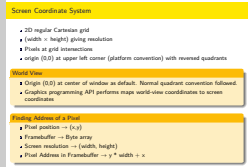
- Origin (0,0) at center of window as default. Normal quadrant convention followed.
- Graphics programming API performs maps world-view coorddinates to screen coordinates

## Finding Address of a Pixel

- Pixel position $\rightarrow$ (x,y)
- Framebuffer $\rightarrow$ Byte array
- Screen resolution $\rightarrow$ (width, height)
- Pixel Address in Framebuffer $\rightarrow$ y * width + x

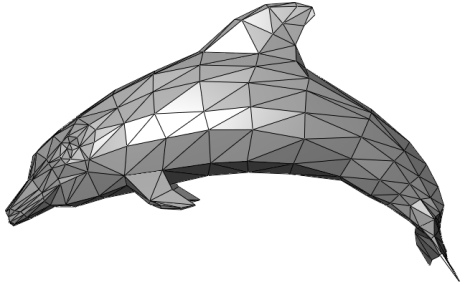




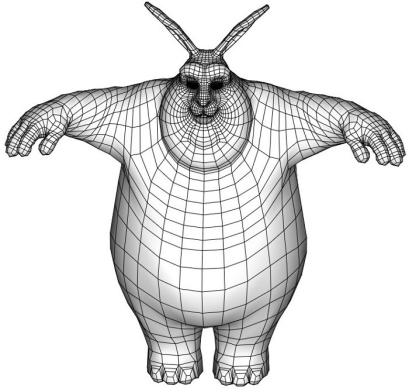


# Meshes

Triangular Mesh

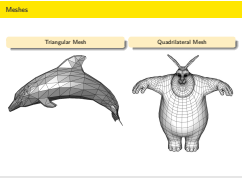


Quadrilateral Mesh









# Issues

## Issues: Quads vs Triangles

- Drawing on a Plane easy with Triangles
- Sub-Division may not result in new vertices
- Simplification by replacement of fine with coarse mesh
- Keep Geometry (Vertices) and Topology (Faces) separate

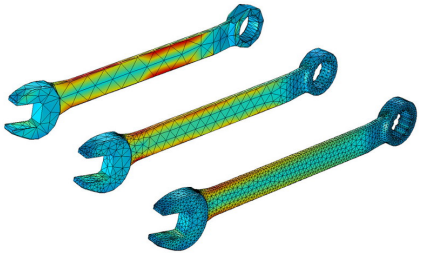


Figure 1: Coarse vs Fine Mesh

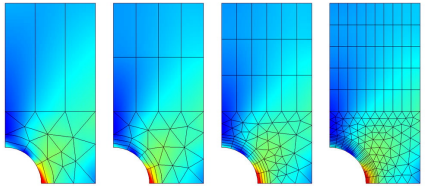
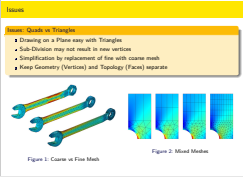


Figure 2: Mixed Meshes







# Surfaces

## Polygon

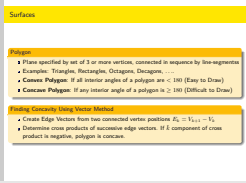
- Plane specified by set of 3 or more vertices, connected in sequence by line-segmentss
- Examples: Triangles, Rectangles, Octagons, Decagons, ....
- **Convex Polygon**: If all interior angles of a polygon are $< 180$ (Easy to Draw)
- **Concave Polygon**: If any interior angle of a polygon is $\geq 180$ (Difficult to Draw)

## Finding Concavity Using Vector Method

- Create Edge Vectors from two connected vertex positions $E_k = V_{k+1} - V_k$
- Determine cross products of successive edge vectors. If $\hat{k}$ component of cross product is negative, polygon is concave.







# Straight Line

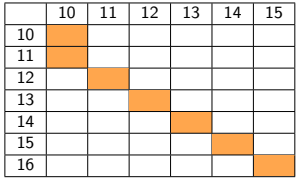
- $y = mx + b$
- $m = \dfrac{y_1 - y_0}{x_1 - x_0}$
- $b = y - mx$

**Rasterization Effects**

```
m = getSlope(p1, p2);
if m ≤ 1 then
    x = x + Δ x
    y = m x + b
else
    y = y + Δ y
    x = (y - b) / m
end if
```

- $p_1 = (10, 10), p_2 = (15, 16)$
- $\implies m = 1.2, b = -2$
- If pixel size is $1 \times 1$, $\implies \Delta x = 1, \Delta y = 1$

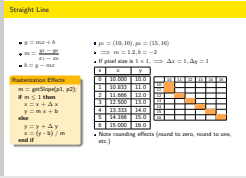
| s | x | y |
|---|---|---|
| 0 | 10.000 | 10.0 |
| 1 | 10.833 | 11.0 |
| 2 | 11.666 | 12.0 |
| 3 | 12.500 | 13.0 |
| 4 | 13.333 | 14.0 |
| 5 | 14.166 | 15.0 |
| 6 | 15.000 | 16.0 |




- Note rounding effects (round to zero, round to one, etc.)







# Circles

- Center$(x_c, y_c)$, Radius $r$, Diameter, Circumference
- Perimeter $C = 2\pi r$
- Known Inputs: center point and radius

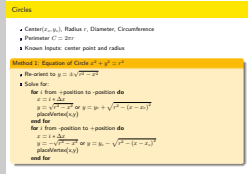
## Method 1: Equation of Circle $x^2 + y^2 = r^2$

- Re-orient to $y = \pm\sqrt{r^2 - x^2}$
- Solve for:

**for** $i$ from +position to -position **do**
  $x = i * \Delta x$
  $y = \sqrt{r^2 - x^2}$ or $y = y_c + \sqrt{r^2 - (x - x_c)^2}$
  placeVertex(x,y)
**end for**
**for** $i$ from -position to +position **do**
  $x = i * \Delta x$
  $y = -\sqrt{r^2 - x^2}$ or $y = y_c - \sqrt{r^2 - (x - x_c)^2}$
  placeVertex(x,y)
**end for**







# Circles (cont.)

## Method 2: Polar Coordinates

- $x = x_c + r\cos\theta$
- $y = y_c + r\sin\theta$
- for loop $\theta = 0 \rightarrow 360$ for degrees
- for loop $i = 0 \rightarrow 2\pi$ for radians





# Extensions to Circles

## Equation of an Ellipse

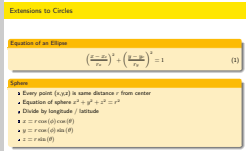
$$\left(\frac{x - x_c}{r_x}\right)^2 + \left(\frac{y - y_c}{r_y}\right)^2 = 1 \quad (1)$$

## Sphere

- Every point (x,y,z) is same distance $r$ from center
- Equation of sphere $x^2 + y^2 + z^2 = r^2$
- Divide by longitude / latitude
- $x = r\cos(\phi)\cos(\theta)$
- $y = r\cos(\phi)\sin(\theta)$
- $z = r\sin(\theta)$







# Graphics Programming Overview

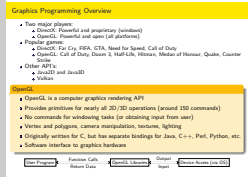
- Two major players:
  - DirectX: Powerful and proprietary (windows)
  - OpenGL: Powerful and open (all platforms)
- Popular games:
  - DirectX: Far Cry, FIFA, GTA, Need for Speed, Call of Duty
  - OpenGL: Call of Duty, Doom 3, Half-Life, Hitman, Medan of Honour, Quake, Counter Strike
- Other API's:
  - Java2D and Java3D
  - Vulkan

## OpenGL

- OpenGL is a computer graphics rendering API
- Provides primitives for nearly all 2D/3D operations (around 150 commands)
- No commands for windowing tasks (or obtaining input from user)
- Vertex and polygons, camera manipulation, textures, lighting
- Originally written for C, but has separate bindings for Java, C++, Perl, Python, etc.
- Software interface to graphics hardware





# OpenGL History

- 1.0 (1992) :: Features in 5 sub-versions
- 2.0 (2004) :: Introduction of Shaders
- 3.0 (2008) :: Changes in the entire Pipeline
- 4.0 (2010) :: Performance (atomic), Architecture changes, OpenGL ES, WebGL ES





OpenGL History
- 1.0 (1992) :: Features in 5 sub-versions
- 2.0 (2004) :: Introduction of Shaders
- 3.0 (2008) :: Changes in the entire Pipeline
- 4.0 (2010) :: Performance (atomic), Architecture changes, OpenGL ES, WebGL ES

# Function Call Categories

- Primitives (Points, Lines, Polygons, Curves, etc.)
- Attributes (Colors, Patterns, etc.)
- Viewing (2D and 3D Camera Placement)
- Transformations (Matrices Translation, Rotation, etc.)
- Input and Output with keyboard, Mouse, or other devices
- Accessing Device and Platform related data





Function Call Categories
- Primitives (Points, Lines, Polygons, Curves, etc.)
- Attributes (Colors, Patterns, etc.)
- Viewing (2D and 3D Camera Placement)
- Transformations (Matrices Translation, Rotation, etc.)
- Input and Output with keyboard, Mouse, or other devices
- Accessing Device and Platform related data

# OpenGL Components

- **GL** The underlying graphics library
- **GLU** OpenGL Utility Library (provides camera movement, image processing/transformation tools)
- **GLUT** OpenGL Utilities Toolkit: Provides window managers, Menus, Callback to display() function (Java uses AWT/Swing instead of GLUT)





OpenGL Components

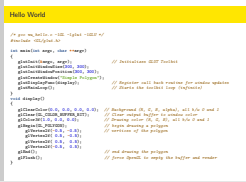
- **GL** The underlying graphics library
- **GLU** OpenGL Utility Library (provides camera movement, image processing/transformation tools)
- **GLUT** OpenGL Utilities Toolkit: Provides window managers, Menus, Callback to display() function (Java uses AWT/Swing instead of GLUT)

# Hello World

```c
/* gcc nu_hello.c -lGL -lglut -lGLU */
#include <GL/glut.h>

int main(int argc, char **argv)
{
   glutInit(&argc, argv);                 // Initializes GLUT Toolkit
   glutInitWindowSize(300, 300);
   glutInitWindowPosition(300, 300);
   glutCreateWindow("Simple Polygon");
   glutDisplayFunc(display);              // Register call back routine for window updates
   glutMainLoop();                        // Starts the toolkit loop (infinite)
}
void display()
{
   glClearColor(0.0, 0.0, 0.0, 0.0);  // Background (R, G, B, alpha), all b/w 0 and 1
   glClear(GL_COLOR_BUFFER_BIT);      // Clear output buffer to window color
   glColor3f(1.0, 0.0, 0.0);          // Drawing color (R, G, B), all b/w 0 and 1
   glBegin(GL_POLYGON);               // begin drawing a polygon
      glVertex2f(-0.5, -0.5);         // vertices of the polygon
      glVertex2f( 0.5, -0.5);
      glVertex2f( 0.5,  0.5);
      glVertex2f(-0.5,  0.5);
   glEnd();                           // end drawing the polygon
   glFlush();                         // force OpenGL to empty the buffer and render
}
```







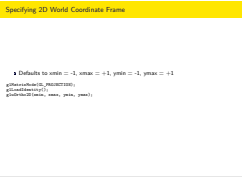
# Specifying 2D World Coordinate Frame

- Defaults to xmin = -1, xmax = +1, ymin = -1, ymax = +1

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluOrtho2D(xmin, xmax, ymin, ymax);
```







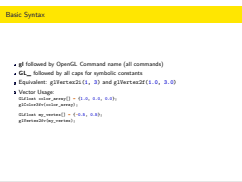
# Basic Syntax

- **gl** followed by OpenGL Command name (all commands)
- **GL_** followed by all caps for symbolic constants
- Equivalent: `glVertex2i(1, 3)` and `glVertex2f(1.0, 3.0)`
- Vector Usage:

```
GLfloat color_array[] = {1.0, 0.0, 0.0};
glColor3fv(color_array);

GLfloat my_vertex[] = {-0.5, 0.5};
glVertex2fv(my_vertex);
```







# Prefixes and Suffixes

| Suffix | Data Type | C Type | OpenGL Type |
|---|---|---|---|
| b | 8-bit int | signed char | GLbyte |
| s | 16-bit int | signed short | GLshort |
| i | 32-bit int | int | GLint |
| f | 32-bit float | float | GLfloat |
| d | 64-bit float | double | GLdouble |
| ub | 8-bit uint | unsigned char | GLubyte |
| us | 16-bit uint | unsigned short | GLushort |
| ui | 32-bit uint | unsigned int | GLuint |





Prefixes and Suffixes

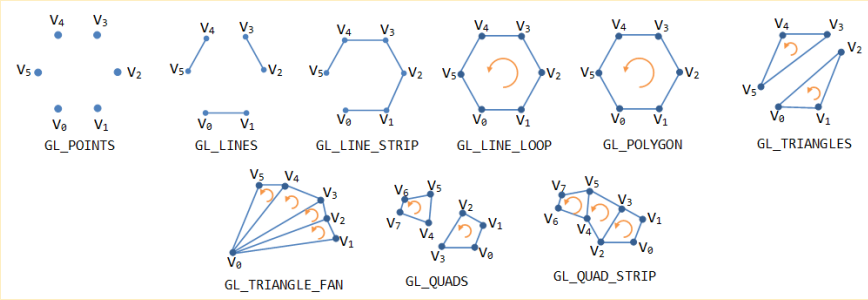
| Suffix | Data Type | C Type | OpenGL Type |
|---|---|---|---|
| b | 8-bit int | signed char | GLbyte |
| s | 16-bit int | signed short | GLshort |
| i | 32-bit int | int | GLint |
| f | 32-bit float | float | GLfloat |
| d | 64-bit float | double | GLdouble |
| ub | 8-bit uint | unsigned char | GLubyte |
| us | 16-bit uint | unsigned short | GLushort |
| ui | 32-bit uint | unsigned int | GLuint |

# Polygon Fill Area Functions

## glRect*(x1, x2, y1, y2)

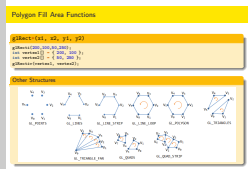
```
glRecti(200,100,50,250);
int vertex1[] = { 200, 100 };
int vertex2[] = { 50, 250 };
glRectiv(vertex1, vertex2);
```

## Other Structures







# Keyboard Special Characters

## glutSpecialFunc(*function) for special keys

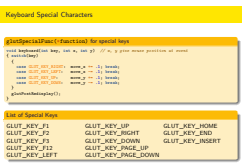
```
void keyboard(int key, int x, int y)  // x, y give mouse position at event
{ switch(key)
  {
    case GLUT_KEY_RIGHT:  move_x += .1; break;
    case GLUT_KEY_LEFT:   move_x -= .1; break;
    case GLUT_KEY_UP:     move_y += .1; break;
    case GLUT_KEY_DOWN:   move_y -= .1; break;
  }
  glutPostRedisplay();
}
```

## List of Special Keys

| | | |
|---|---|---|
| GLUT_KEY_F1 | GLUT_KEY_UP | GLUT_KEY_HOME |
| GLUT_KEY_F2 | GLUT_KEY_RIGHT | GLUT_KEY_END |
| GLUT_KEY_F3 | GLUT_KEY_DOWN | GLUT_KEY_INSERT |
| GLUT_KEY_F12 | GLUT_KEY_PAGE_UP | |
| GLUT_KEY_LEFT | GLUT_KEY_PAGE_DOWN | |







# Keyboard ASCII Characters

## glutKeyboardFunc(*function) for ASCII keys

```
void keyboard(unsigned char key, int x, int y) // x, y give mouse position at event
{ switch(key)
  { // No special key for SHIFT
    case 'A':  case 'a':   move_x -= 1; break;
    case 'S':  case 's':   move_y -= 1; break;
    case 'D':  case 'd':   move_x += 1; break;
    case 'W':  case 'w':   move_y += 1; break;
  }
  glutPostRedisplay();
}
```





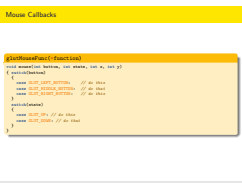
# Mouse Callbacks

## glutMouseFunc(*function)

```
void mouse(int button, int state, int x, int y)
{ switch(button)
  {
    case GLUT_LEFT_BUTTON:    // do this
    case GLUT_MIDDLE_BUTTON:  // do that
    case GLUT_RIGHT_BUTTON:   // do this
  }
  switch(state)
  {
    case GLUT_UP: // do this
    case GLUT_DOWN: // do that
  }
}
```







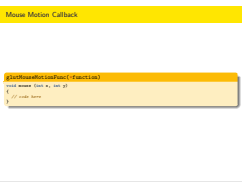
# Mouse Motion Callback

**glutMouseMotionFunc(*function)**

```
void mouse (int x, int y)
{
  // code here
}
```







# Timer Function

**glutTimerFunc(1000/25., timer, 0)**

```
void timer(int x)
{
  rotate_x = x + 0.1;
  rotate_y = x - 0.1;
  glutPostRedisplay();
  glutTimerFunc(1000/25., timer, 0);
}
```





# Storing Coordinate Points as Vector

- Save vertex coordinates as double-scripted array:

```
GLfloat points[8][3] = {  {0, 0,  0}, {.5, 0,  0}, {.5, .5,  0}, {0, .5,  0},
                          {0, 0, .5}, {.5, 0, .5}, {.5, .5, .5}, {0, .5, .5} };
```

- Pass each point to shape generation function, as/when required:

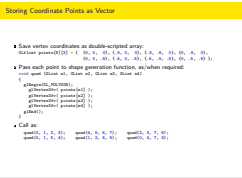
```
void quad (GLint n1, GLint n2, GLint n3, GLint n4)
{
  glBegin(GL_POLYGON);
    glVertex3fv( points[n1] );
    glVertex3fv( points[n2] );
    glVertex3fv( points[n3] );
    glVertex3fv( points[n4] );
  glEnd();
}
```

- Call as:

```
quad(0, 1, 2, 3);    quad(4, 5, 6, 7);    quad(2, 3, 7, 6);
quad(0, 1, 5, 4);    quad(1, 2, 6, 5);    quad(0, 4, 7, 3);
```







# Reduce Number of GL Calls

- Store points as single scripted array

```
GLfloat points[] = {   0,  0,  0,     .5,  0,  0,     .5, .5,  0,     0, .5,  0,
                       0,  0, .5,     .5,  0, .5,     .5, .5, .5,     0, .5, .5 };
```

- Enable Client side Storage

```
glEnableClientState(GL_VERTEX_ARRAY);
glDisableClientState(GL_VERTEX_ARRAY)
```

- Link points to the Vertex Array

```
glVertexPointer(3, GL_FLOAT, 0, points);
```

- Which points make surfaces

```
GLubyte vertIndex = { 0, 1, 2, 3,          4, 5, 6, 7,
                      2, 3, 7, 6,          0, 1, 5, 4,
                      1, 2, 6, 5,          0, 4, 7, 3 };
```

- Draw elements based on vertex indices

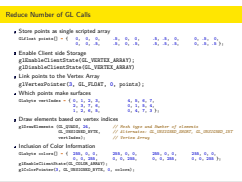
```
glDrawElements (GL_QUADS, 24,          // Mesh type and Number of elements
                GL_UNSIGNED_BYTE,      // Alternates: GL_UNSIGNED_SHORT, GL_UNSIGNED_INT
                vertIndex);            // Vertex Array
```

- Inclusion of Color Information

```
GLubyte colors[] = {  255, 0, 0,     255, 0, 0,     255, 0, 0,     255, 0, 0,
                      0, 0, 255,     0, 0, 255,     0, 0, 255,     0, 0, 255 };
glEnableClientState(GL_COLOR_ARRAY);
glColorPointer(3, GL_UNSIGNED_BYTE, 0, colors);
```







# GLUT Functions for Polygons

- Wire structures vs Solid Structure
- Four sided Tetrahedron (Pyramid) `glutWireTetrahedron()` or `glutSolidTetrahedron()`
- Six sided Regular Hexahedron (Cube) `glutWireCube(edgeLength)` or `glutSolidCube(edgeLength)}`
- Eight sided Octahedron `glutWireOctahedron()` or `glutSolidOctahedron`
- Twelve sided Dodecahedron `glutWireDodecahedron()` or `glutSolidDodecahedron()`
- Twenty sided Icosahedron `glutWireIcosahedron()` or `glutSolidIcosahedron()`
- Center of all objects frawn at origin world coordinates.





GLUT Functions for Polygons

- Wire structures vs Solid Structure
- Four sided Tetrahedron (Pyramid) glutWireTetrahedron() or glutSolidTetrahedron()
- Six sided Regular Hexahedron (Cube) glutWireCube(edgeLength) or glutSolidCube(edgeLength)}
- Eight sided Octahedron glutWireOctahedron() or glutSolidOctahedron
- Twelve sided Dodecahedron glutWireDodecahedron() or glutSolidDodecahedron()
- Twenty sided Icosahedron glutWireIcosahedron() or glutSolidIcosahedron()
- Center of all objects frawn at origin world coordinates.

# GLUT functions for Curved Surfaces

- `glutWireSphere(radius, longitude, latitude)`
- `glutWireCone(base, height, longitude, latitude)`
- `glutWireTorus(rCrossSection, rAxial, nConcentric, nRadialSlices)`
- `glutWireTeapot(size)`





GLUT functions for Curved Surfaces
- glutWireSphere(radius, longitude, latitude)
- glutWireCone(base, height, longitude, latitude)
- glutWireTorus(rCrossSection, rAxial, nConcentric, nRadialSlices)
- glutWireTeapot(size)

# Texture Overview

- Mapping of 2D Image to 3D Object
- `Image[height][weight][4]`
- Texture Coordinates bounded in $[0, 1]$. OpenGL uses $(s, t)$ to refer to them, others use $(u, v)$.
- Pixels of textures called *texels*
- Mapping Operation: Takes 3D points to (u,v) coordinates (Easy for cubes, spheres, complicated for others)

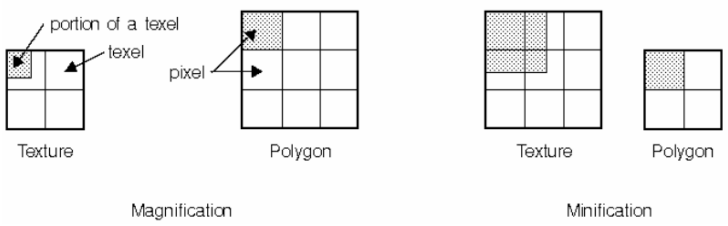
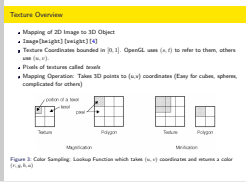



Figure 3: Color Sampling: Lookup Function which takes $(u, v)$ coordinates and returns a color $(r, g, b, a)$







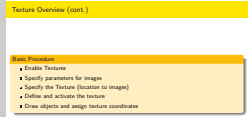
# Texture Overview (cont.)

## Basic Procedure

- Enable Textures
- Specify parameters for images
- Specify the Texture (location to images)
- Define and activate the texture
- Draw objects and assign texture coordinates







# Texture Enabling

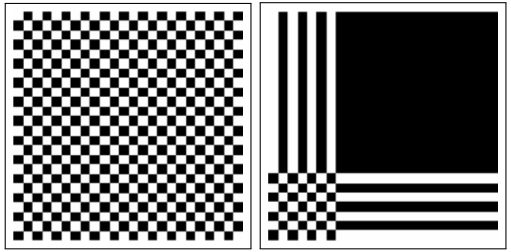
- Enabling of Textures
  - `glEnable(GL_TEXTURE_2D)`
  - `glDisable(GL_Texture_2D)`
- Create a Texture Object
  - `glGenTextures(1, &texture_id)`
  - Link: `glBindTexture(GL_TEXTURE_2D, texture_id)`
  - Unlink: `glBindTexture(0)`





# Texture Parameters

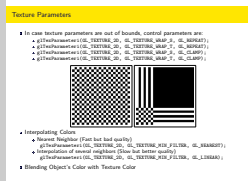
- In case texture parameters are out of bounds, control parameters are:
  - `glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);`
  - `glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);`
  - `glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP);`
  - `glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP);`



- Interpolating Colors
  - Nearest Neighbor (Fast but bad quality)
    `glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);`
  - Interpolation of several neighbors (Slow but better quality)
    `glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);`
- Blending Object's Color with Texture Color





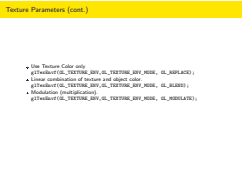


# Texture Parameters (cont.)

- Use Texture Color only
  `glTexEnvf(GL_TEXTURE_ENV,GL_TEXTURE_ENV_MODE, GL_REPLACE);`
- Linear combination of texture and object color.
  `glTexEnvf(GL_TEXTURE_ENV,GL_TEXTURE_ENV_MODE, GL_BLEND);`
- Modulation (multiplication).
  `glTexEnvf(GL_TEXTURE_ENV,GL_TEXTURE_ENV_MODE, GL_MODULATE);`







# Texture Location an Activation

## Texture Location

- Nothing in OpenGL to load an image !!!!!!!!
- 3rd Party APIS
  - Simple OpenGL Image Library (Link: lonesock.net/soil.html)
  - Single File Public Domain Libraries for C/C++ (Link: github.com/nothings/stb)
- `GLubyte *data = stbi_load("img.jpg", &width, &height, &channels, 0);`
  - Channel: Number of 8-bit components per pixel.
  - Param 5: Force different density on Channel.





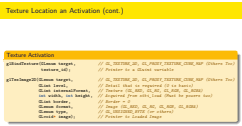
# Texture Location an Activation (cont.)

## Texture Activation

```
glBindTexture(GLenum target,          // GL_TEXTURE_2D, GL_PROXY_TEXTURE_CUBE_MAP (Others Too)
              texture_id);            // Pointer to a GLuint variable

glTexImage2D(GLenum target,           // GL_TEXTURE_2D, GL_PROXY_TEXTURE_CUBE_MAP (Others Too)
             GLint level,             // Detail that is required (0 is basic)
             GLint internalFormat,    // Texture (GL_RED, GL_RG, GL_RGB, GL_RGBA)
             int width, int height,   // Acquired from stbi_load (Must be powers two)
             GLint border,            // Border = 0
             GLenum format,           // Image (GL_RED, GL_RG, GL_RGB, GL_RGBA)
             GLenum type,             // GL_UNSIGNED_BYTE (or others)
             GLvoid* image);          // Pointer to Loaded Image
```







# Draw Objects and Assign Textures

```
glEnable(GL_TEXTURE_2D);
glBindTexture(GL_TEXTURE_2D, texture_id);

glBegin(GL_POLYGON);
  glTexCoord2f(0.0, 0.0);        glVertex3f( 0,   0,   0 );
  glTexCoord2f(1.0, 0.0);        glVertex3f( 0.5, 0,   0 );
  glTexCoord2f(1.0, 1.0);        glVertex3f( 0.5, 0.5, 0 );
  glTexCoord2f(0.0, 1.0);        glVertex3f( 0,   0.5, 0 );
glEnd();
glBindTexture(GL_TEXTURE_2D, 0);
glDisable(GL_TEXTURE_2D);
```





Draw Objects and Assign Textures

glEnable(GL_TEXTURE_2D);
glBindTexture(GL_TEXTURE_2D, texture_id);

glBegin(GL_POLYGON);
  glTexCoord2f(0.0, 0.0);  glVertex3f( 0,  0,  0 );
  glTexCoord2f(1.0, 0.0);  glVertex3f( 0.5, 0,  0 );
  glTexCoord2f(1.0, 1.0);  glVertex3f( 0.5, 0.5, 0 );
  glTexCoord2f(0.0, 1.0);  glVertex3f( 0,  0.5, 0 );
glEnd();
glBindTexture(GL_TEXTURE_2D, 0);
glDisable(GL_TEXTURE_2D);

# Lighting and Illumination

- Enabling Lighting: `glEnable(GL_LIGHTING)`
- Create Point Light Sources :

```
GLfloat white[] = {1.0, 1.0, 1.0, 1.0};
glLightfv(Name,         // Light Source Name (GL_LIGHT0, GL_LIGHT1, ..., GL_LIGHT7)
          Property,     // Light Property (GL_AMBIENT, GL_DIFFUSE, GL_SPECULAR)
          Pointer);     // in this case white
```

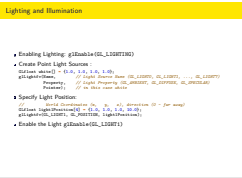
- Specify Light Position:

```
//          World Coordinates (x,   y,   z), direction (0 - far away)
GLfloat light1Position[4] = {1.0, 1.0, 1.0, 10.0};
glLightfv(GL_LIGHT1, GL_POSITION, light1Position);
```

- Enable the Light `glEnable(GL_LIGHT1)`







1. Misc. Topics
   - Pixel Primitives
   - Object Management
   - View/Display

# 2D Geometric Transforms

- Typical Transformations (Linear): Translation, Rotation, Scaling
- Every point in object is moved same distance (without deformation)
- For complicated objects, translate basis point, and redraw other points relative to it

## Translation

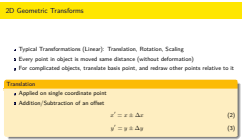
- Applied on single coordinate point
- Addition/Subtraction of an offset

$$x' = x \pm \Delta x \tag{2}$$

$$y' = y \pm \Delta y \tag{3}$$







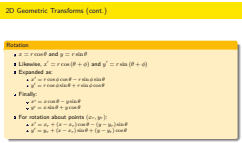
# 2D Geometric Transforms (cont.)

## Rotation

- $x = r\cos\theta$ and $y = r\sin\theta$
- Likewise, $x' = r\cos(\theta + \phi)$ and $y' = r\sin(\theta + \phi)$
- Expanded as:
  - $x' = r\cos\phi\cos\theta - r\sin\phi\sin\theta$
  - $y' = r\cos\phi\sin\theta + r\sin\phi\cos\theta$
- Finally:
  - $x' = x\cos\theta - y\sin\theta$
  - $y' = x\sin\theta + y\cos\theta$
- For rotation about points $(x_r, y_r)$:
  - $x' = x_r + (x - x_r)\cos\theta - (y - y_r)\sin\theta$
  - $y' = y_r + (x - x_r)\sin\theta + (y - y_r)\cos\theta$







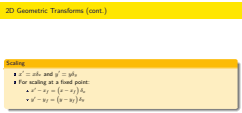
# 2D Geometric Transforms (cont.)

## Scaling

- $x' = x\delta_x$ and $y' = y\delta_y$
- For scaling at a fixed point:
  - $x' - x_f = \left(x - x_f\right)\delta_x$
  - $y' - y_f = \left(y - y_f\right)\delta_y$







# Transformations in OpenGL

- **Translation**: 4x4 matrix generated by `glTranslatef(x,y,z)` and applied to all vertices
- **Rotation**: 4x4 matrix generated by `glRotatef(theta,x,y,z)` and applied to all vertices
- **Scaling**: 4x4 matrix generated by `glScalef(x,y,z)` and applied to all vertices





Transformations in OpenGL

- **Translation**: 4x4 matrix generated by `glTranslatef(x,y,z)` and applied to all vertices
- **Rotation**: 4x4 matrix generated by `glRotatef(theta,x,y,z)` and applied to all vertices
- **Scaling**: 4x4 matrix generated by `glScalef(x,y,z)` and applied to all vertices

# Homogeneous Coordinate System

$$\mathbf{P}' = M_1\mathbf{P} + M_2 \tag{4}$$

- $\mathbf{P}$ is cartesian coordinates $(x, y)$
- Homogeneous coordinate system aims to represent all transformations as matrix multiplications

$$\mathbf{P}'_h = M_c\mathbf{P}_h \tag{5}$$

- $\mathbf{P}_h$ is homogeneous coordinates $(x_h, y_h, h)$
- $x = x_h/h$, and $y = y_h/h$
- For convenience, $h = 1$

**Translation**

$$\begin{bmatrix} 1 & 0 & \Delta x \\ 0 & 1 & \Delta y \\ 0 & 0 & 1 \end{bmatrix}$$

**Rotation**

$$\begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

**Scaling**

$$\begin{bmatrix} \delta_x & 0 & 0 \\ 0 & \delta_y & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

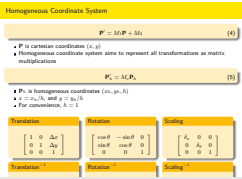
**Translation$^{-1}$**

**Rotation$^{-1}$**

**Scaling$^{-1}$**







# Other Homogeneous Coordinate Transforms

**Reflection in x**

$$\begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

**Reflection in y**

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

**Shear along x**

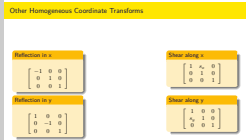
$$\begin{bmatrix} 1 & s_x & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

**Shear along y**

$$\begin{bmatrix} 1 & 0 & 0 \\ s_y & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$







# Matrix Operations

## Basic Matrix Mode

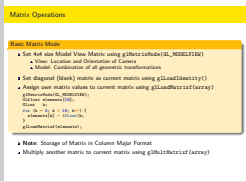
- Set 4x4 size Model View Matrix using `glMatrixMode(GL_MODELVIEW)`
  - View: Location and Orientation of Camera
  - Model: Combination of all geometric transformations
- Set diagonal (blank) matrix as current matrix using `glLoadIdentity()`
- Assign own matrix values to current matrix using `glLoadMatrixf(array)`

```
glMatrixMode(GL_MODELVIEW);
GLfloat elements[16];
GLint   k;
for (k = 0; k < 16; k++) {
  elements[k] = (float)k;
}
glLoadMatrixf(elements);
```

- **Note**: Storage of Matrix in Column Major Format
- Multiply another matrix to current matrix using `glMultMatrixf(array)`

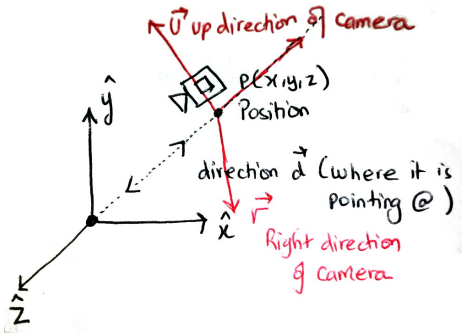






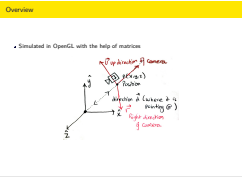
# Overview

- Simulated in OpenGL with the help of matrices







# Overview (cont.)

## GL_MODEL_VIEW Matrix: Position and Orientation of Camera and World

Modelview position and orientation of camera.

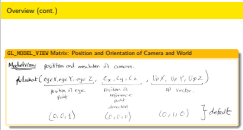
gluLookat( eyeX, eyeY, eyeZ, Cx, Cy, Cz, UpX, UpY, UpZ )

- eyeX, eyeY, eyeZ: position of eye point
- Cx, Cy, Cz: position of reference point / direction
- UpX, UpY, UpZ: up vector.

(0, 0, 1) (0, 0, 0) (0, 1, 0) } default

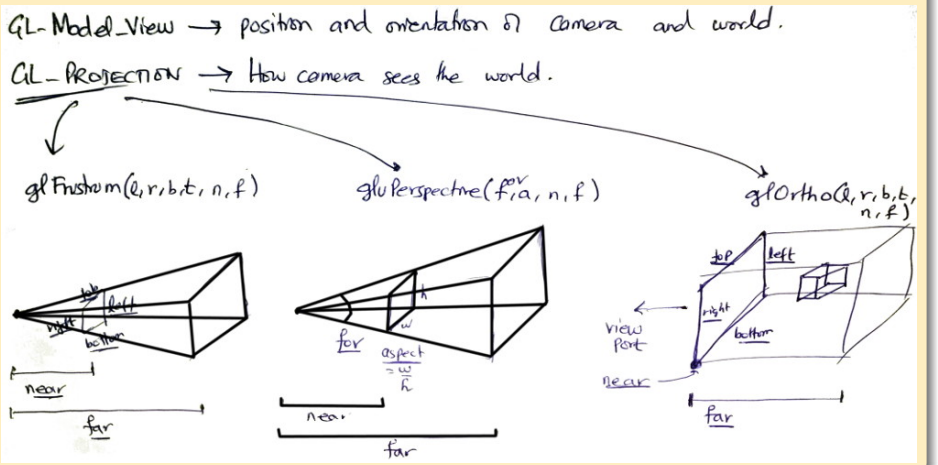






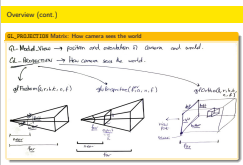
# Overview (cont.)

## GL_PROJECTION Matrix: How camera sees the world







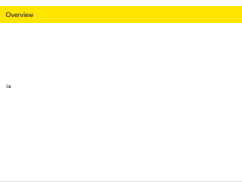



# Overview

ia







# Overview

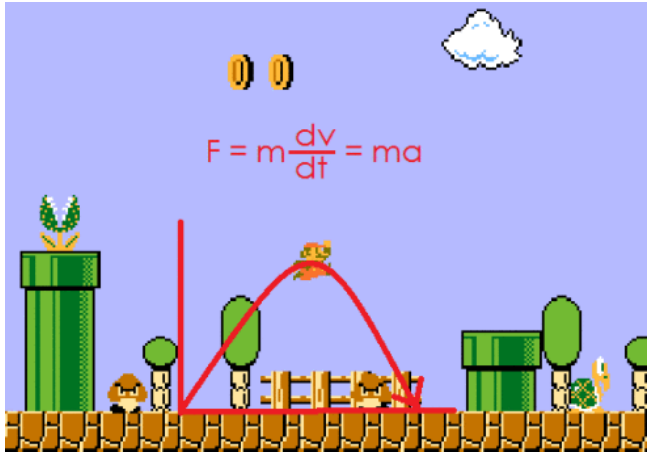



Figure 4: Physics in Super-Mario





Overview

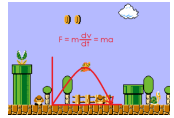
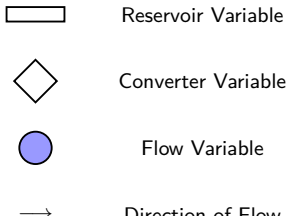


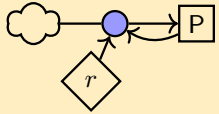

Figure 4: Physics in Super-Mario

# Diagrammatic Representation




## Population Growth Example

$$\frac{\mathrm{d}P}{\mathrm{d}t} = rP \qquad (6)$$

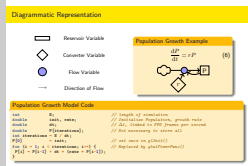



## Population Growth Model Code

```
int             N;                      // length of simulation
double          init, rate;             // Initialize Population, growth rate
double          dt;                     // Δt, linked to FPS frames per second
double          P[iterations];          // Not necessary to store all
int iterations = N / dt;
P[0]            = init;                 // set once in glInit()
for (i = 1; i < iterations; i++) {      // Replaced by glutTimerFunc()
 P[i] = P[i-1] + dt * (rate * P[i-1]);
}
```



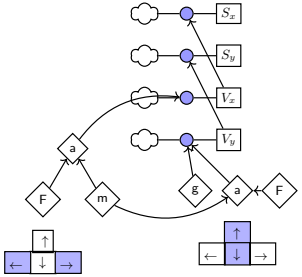


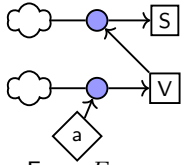


# Base Model for Motion

$$\text{Velocity } v = \frac{\mathrm{d}s}{\mathrm{d}t} \qquad (7)$$

$$\text{Acceleration } a = \frac{\mathrm{d}v}{\mathrm{d}t} \qquad (8)$$

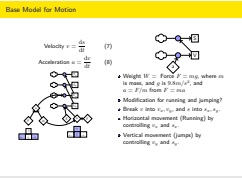



- Weight $W =$ Force $F = mg$, where $m$ is mass, and $g$ is $9.8m/s^2$, and $a = F/m$ from $F = ma$
- Modification for running and jumping?
- Break $v$ into $v_x, v_y$, and $s$ into $s_x, s_y$.
- Horizontal movement (Running) by controlling $v_x$ and $s_x$.
- Vertical movement (jumps) by controlling $v_y$ and $s_y$.

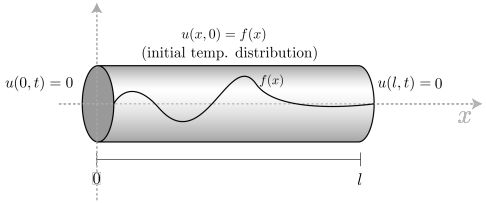






# 1D Heat Equation

$$\frac{\partial \mathbf{U}}{\partial t} = \frac{k}{c\rho}\frac{\partial^2 \mathbf{U}}{\partial x^2} \qquad (9)$$

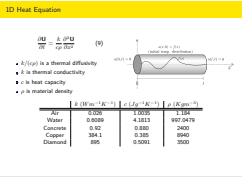



- $k/(c\rho)$ is a thermal diffusivity
- $k$ is thermal conductivity
- $c$ is heat capacity
- $\rho$ is material density

| | $k\ (Wm^{-1}K^{-1})$ | $c\ (Jg^{-1}K^{-1})$ | $\rho\ (Kgm^{-3})$ |
|---|---|---|---|
| Air | 0.026 | 1.0035 | 1.184 |
| Water | 0.6089 | 4.1813 | 997.0479 |
| Concrete | 0.92 | 0.880 | 2400 |
| Copper | 384.1 | 0.385 | 8940 |
| Diamond | 895 | 0.5091 | 3500 |







# Realization into OpenGL

## Display Code

```
glClearColor(0.0, 0.0, 0.0, 0.0);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
glEnable(GL_DEPTH_TEST);

glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glOrtho(-5.,5.,-5,5.,-5.,5.);

glRotatef( rotate_y, 1.0, 0.0, 0.0 );
glRotatef( rotate_x, 0.0, 1.0, 0.0 );

cubeMesh(-2, +2., -2, 2, -2, 2, 10, 10, 10);

glFlush();
glutSwapBuffers();
```





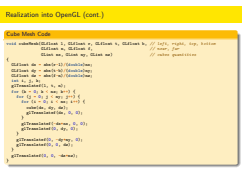
# Realization into OpenGL (cont.)

## Cube Mesh Code

```
void cubeMesh(GLfloat l, GLfloat r, GLfloat t, GLfloat b, // left, right, top, bottom
              GLfloat n, GLfloat f,                       // near, far
              GLint nx, GLint ny, GLint nz)               // cubes quantities
{
  GLfloat dx = abs(r-l)/(double)nx;
  GLfloat dy = abs(t-b)/(double)ny;
  GLfloat dz = abs(f-n)/(double)nz;
  int i, j, k;
  glTranslatef(l, t, n);
  for (k = 0; k < nz; k++) {
    for (j = 0; j < ny; j++) {
      for (i = 0; i < nx; i++) {
        cube(dx, dy, dz);
        glTranslatef(dx, 0, 0);
      }
      glTranslatef(-dx*nx, 0, 0);
      glTranslatef(0, dy, 0);
    }
    glTranslatef(0, -dy*ny, 0);
    glTranslatef(0, 0, dz);
  }
  glTranslatef(0, 0, -dz*nz);
}
```







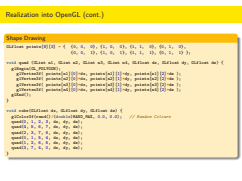
# Realization into OpenGL (cont.)

## Shape Drawing

```
GLfloat points[8][3] = {  {0, 0,  0}, {1, 0,  0}, {1, 1,  0}, {0, 1,  0},
                          {0, 0,  1}, {1, 0,  1}, {1, 1,  1}, {0, 1,  1} };

void quad (GLint n1, GLint n2, GLint n3, GLint n4, GLfloat dx, GLfloat dy, GLfloat dz) {
  glBegin(GL_POLYGON);
    glVertex3f( points[n1][0]*dx, points[n1][1]*dy, points[n1][2]*dz );
    glVertex3f( points[n2][0]*dx, points[n2][1]*dy, points[n2][2]*dz );
    glVertex3f( points[n3][0]*dx, points[n3][1]*dy, points[n3][2]*dz );
    glVertex3f( points[n4][0]*dx, points[n4][1]*dy, points[n4][2]*dz );
  glEnd();
}

void cube(GLfloat dx, GLfloat dy, GLfloat dz) {
  glColor3f(rand()/(double)RAND_MAX, 0.0, 0.0);   // Random Colours
  quad(0, 1, 2, 3, dx, dy, dz);
  quad(4, 5, 6, 7, dx, dy, dz);
  quad(2, 3, 7, 6, dx, dy, dz);
  quad(0, 1, 5, 4, dx, dy, dz);
  quad(1, 2, 6, 5, dx, dy, dz);
  quad(3, 7, 4, 0, dx, dy, dz);
}
```







# Display

## Sample Videos

Random Heat

(Click)

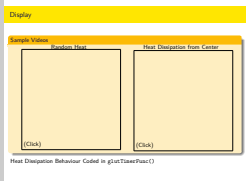
Heat Dissipation from Center

(Click)

Heat Dissipation Behaviour Coded in `glutTimerFunc()`







# Shaders (Non-Examinable)

- Small program(s) that rest and run on the GPU (and control parts of the **graphics pipeline** such as Vertex, Geometry, and Fragment Shaders)
- Compilation, Linking, and Loading in OpenGL itself.
- Programmed using the **OpenGL Shading Language** (C like language tailored for graphics, and has features specialized for vector and matrix operations)

## Vertex Shaders

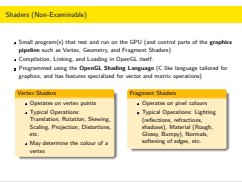
- Operates on vertex points
- Typical Operations: Translation, Rotation, Skewing, Scaling, Projection, Distortions, etc.
- May determine the colour of a vertex

## Fragment Shaders

- Operates on pixel colours
- Typical Operations: Lighting (reflections, refractions, shadows), Material (Rough, Glossy, Bumpy), Normals, softening of edges, etc.







# General Syntax

```
#version version_number

flag type VariableName;

void main()
{
    variable_name =   // do something;
}
```

- **Version**: Typically 120 for version 1.20
- **flag**: Can be in, out, inout, attribute, varying, or uniform
  - **In**: Values passed into a function
  - **Out**: Values passed out of a function
  - **Attribute**: Link to vertex attributes (data associated with each vertex), so passed to vertex processor only
  - **Varying**: Passed from vertex processor to fragment processor
  - **Uniform**: Has global scope (does not change contents while changing from one shader program to another). In a way, serves as constants, accessible to both vertex and fragment shaders.





General Syntax

#version version_number

flag type VariableName;

void main()
{
variable_name = // do something;
}

- Version: Typically 120 for version 1.20
- flag: Can be in, out, inout, attribute, varying, or uniform
  - In: Values passed into a function
  - Out: Values passed out of a function
  - Attribute: Link to vertex attributes (data associated with each vertex), so passed to vertex processor only
  - Varying: Passed from vertex processor to fragment processor
  - Uniform: Has global scope (does not change contents while changing from one shader program to another). In a way, serves as constants, accessible to both vertex and fragment shaders.

# General Syntax (cont.)

- **Type**: Can be conventional data types (int, float, etc.), or vector data types (intn, floatn, etc.), or matrix data types, (mat4), or aggregate data types (structs), or arrays

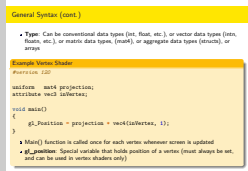
## Example Vertex Shader

```
#version 120

uniform   mat4 projection;
attribute vec3 inVertex;

void main()
{
     gl_Position = projection * vec4(inVertex, 1);
}
```

- Main() function is called once for each vertex whenever screen is updated
- **gl_position**: Special variable that holds position of a vertex (must always be set, and can be used in vertex shaders only)







# General Syntax (cont.)

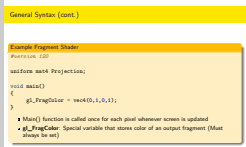
## Example Fragment Shader

```
#version 120

uniform mat4 Projection;

void main()
{
     gl_FragColor = vec4(0,1,0,1);
}
```

- Main() function is called once for each pixel whenever screen is updated
- **gl_FragColor**: Special variable that stores color of an output fragment (Must always be set)







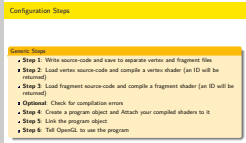
# Configuration Steps

## Generic Steps

- **Step 1**: Write source-code and save to separate vertex and fragment files
- **Step 2**: Load vertex source-code and compile a vertex shader (an ID will be returned)
- **Step 3**: Load fragment source-code and compile a fragment shader (an ID will be returned)
- **Optional**: Check for compilation errors
- **Step 4**: Create a program object and Attach your compiled shaders to it
- **Step 5**: Link the program object
- **Step 6**: Tell OpenGL to use the program







# Configuration Steps (cont.)

**Step 1**: Write your source code and save to files

- Our Vertex Shader

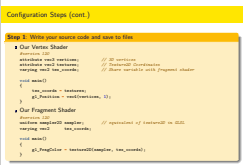
```
#version 120
attribute vec3 vertices;          // 3D vertices
attribute vec2 textures;          // Texture2D Coordinates
varying vec2 tex_coords;          // Share variable with fragment shader

void main()
{
     tex_coords = textures;
     gl_Position = vec4(vertices, 1);
}
```

- Our Fragment Shader

```
#version 120
uniform sampler2D sampler;        // equivalent of texture2D in GLSL
varying vec2      tex_coords;

void main()
{
     gl_FragColor = texture2D(sampler, tex_coords);
}
```







# Configuration Steps (cont.)

**Step 2**: Load vertex source-code and compile vertex shader

```
int vs = glCreateShader(GL_VERTEX_SHADER);
glShaderSource(vs, readFile(vShader));
glCompileShader(vs);
```

**Step 3**: Load Fragment source-code and compile fragment shader

```
int fs = glCreateShader(GL_FRAGMENT_SHADER);
glShaderSource(fs, readFile(fShader));
glCompileShader(fs);
```





Configuration Steps (cont.)
Step 2: Load vertex source-code and compile vertex shader
int vs = glCreateShader(GL_VERTEX_SHADER);
glShaderSource(vs, readFile(vShader));
glCompileShader(vs);
Step 3: Load Fragment source-code and compile fragment shader
int fs = glCreateShader(GL_FRAGMENT_SHADER);
glShaderSource(fs, readFile(fShader));
glCompileShader(fs);

# Configuration Steps (cont.)

**Optional**: Checking for Errors after Compilation

```
if(glGetShaderi(vs, GL_COMPILE_STATUS) != 1) {
                printf("%s\n", glGetShaderInfoLog(fs));
}

if(glGetShaderi(fs, GL_COMPILE_STATUS) != 1) {
                printf("%s\n", glGetShaderInfoLog(fs));
}
```

**Step 4**: Create a Program Object and Attach Shaders

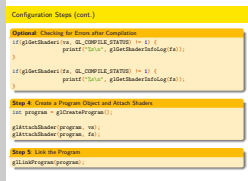
```
int program = glCreateProgram();

glAttachShader(program, vs);
glAttachShader(program, fs);
```

**Step 5**: Link the Program

```
glLinkProgram(program);
```







# Configuration Steps (cont.)

Use the following if you want to check status of linking:

```
if(glGetProgrami(program, GL_LINK_STATUS) != 1) {
                printf("%s\n", glGetProgramInfoLog(program));
}
```

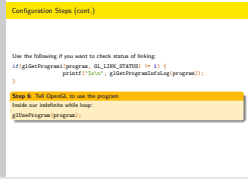
**Step 6**: Tell OpenGL to use the program

Inside our indefinite while loop:

```
glUseProgram(program);
```







# Pixel Level Primitives

- Fill on Raster Positions using `glDrawPixels()` or `glBitmap()`

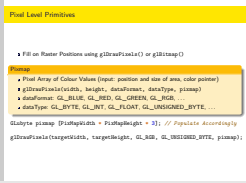
## Pixmap

- Pixel Array of Colour Values (input: position and size of area, color pointer)
- `glDrawPixels(width, height, dataFormat, dataType, pixmap)`
- dataFormat: GL_BLUE, GL_RED, GL_GREEN, GL_RGB, ...
- dataType: GL_BYTE, GL_INT, GL_FLOAT, GL_UNSIGNED_BYTE, ...

```
GLubyte pixmap [PixMapWidth * PixMapHeight * 3]; // Populate Accordingly

glDrawPixels(targetWidth, targetHeight, GL_RGB, GL_UNSIGNED_BYTE, pixmap);
```







# Pixel Level Primitives (cont.)

## Bitmap Masks

- Assignment of Bit 0 or 1 to each element of a matrix (binary image)
- `glBitmap(width, height, offX, offY, screenX, screenY, bitShape)`
- offX, offY: Offset in Memory Location
- screenX, screenY: Offset on Screen (pixels)

```
GLubyte bitShape [20] = { 0x1c, 0x00, 0x1c, 0x00, 0x1c, 0x00,
                          0x1c, 0x00, 0x1c, 0x00, 0xff, 0x80,
                          0x7f, 0x00, 0x3e, 0x00, 0x1c, 0x00, 0x08, 0x00 }:

glPixelStorei (GL_UNPACK_ALIGNMENT. 1): // Set pixel storage mods
glBitmap (9. 10. 0.0. 0.0. 20.0. 15.0, bitShape);
```

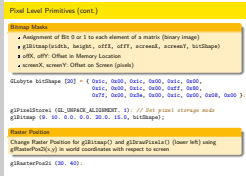
## Raster Position

Change Raster Position for `glBitmap()` and `glDrawPixels()` (lower left) using glRasterPos2i(x,y) in world coordinates with respect to screen

```
glRasterPos2i (30. 40):
```







# Pixel Level Primitives (cont.)

## Read Frame Buffer Pixels

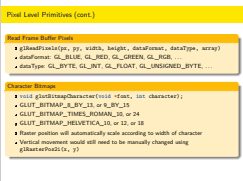
- `glReadPixels(px, py, width, height, dataFormat, dataType, array)`
- dataFormat: GL_BLUE, GL_RED, GL_GREEN, GL_RGB, ...
- dataType: GL_BYTE, GL_INT, GL_FLOAT, GL_UNSIGNED_BYTE, ...

## Character Bitmaps

- `void glutBitmapCharacter(void *font, int character);`
- GLUT_BITMAP_8_BY_13, or 9_BY_15
- GLUT_BITMAP_TIMES_ROMAN_10, or 24
- GLUT_BITMAP_HELVETICA_10, or 12, or 18
- Raster position will automatically scale according to width of character
- Vertical movement would still need to be manually changed using `glRasterPos2i(x, y)`







# Object Management using Display Lists

## Display Lists

- Identify Objects as named entities (in OpenGL environment)
- Handler/Shortcut to Object
- Enclose object construction commands within:

```
glNewList(listID, listMode);
...
glEndList();
```

- listID: Positive integer
- listMode: GL_COMPILE, GL_COMPILE_AND_EXECUTE

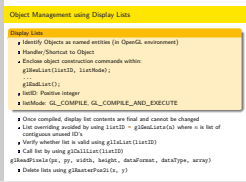
- Once compiled, display list contents are final and cannot be changed
- List overriding avoided by using `listID = glGenLists(n)` where $n$ is list of contiguous unused ID's
- Verify whether list is valid using `glIsList(listID)`
- Call list by using `glCallList(listID)`

```
glReadPixels(px, py, width, height, dataFormat, dataType, array)
```

- Delete lists using `glRasterPos2i(x, y)`







# View/Display Modification

## Clipping Window

A 2D scene that is selected for display (where all objects out of this scene are clipped)

## Viewport

- Objects identified in clipping window are mapped to viewports
- `glViewport(startx, starty, endx, endy)`
- Has it's own world coordinates