XIAMEN UNIVERSITY

COMPUTER GRAPHICS

An OpenGL Toolbox

Dr. Zhonggui Chen
School of Informatics, Xiamen University

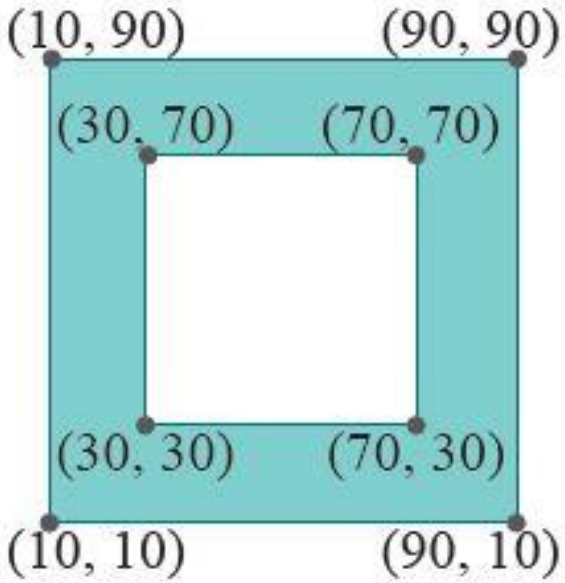
http://graphics.xmu.edu.cn

OpenGL Toolbox

Vertex Arrays and
Their Drawing Commands

\Chapter3\squareAnnulus1.cpp

- a plain-vanilla program which draws the square annulus

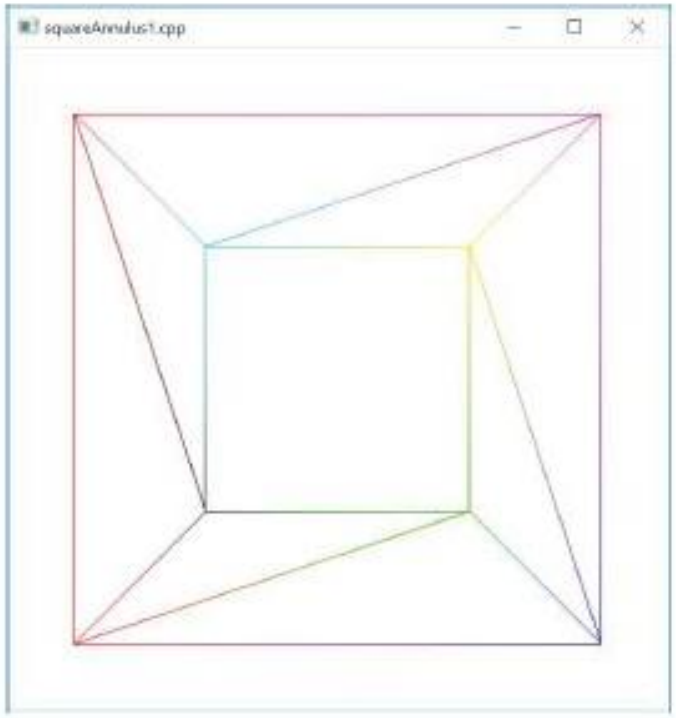



```
glBegin(GL_TRIANGLE_STRIP);
   glColor3f(0.0, 0.0, 0.0);
   glVertex3f(30.0, 30.0, 0.0); // Vertex 0
   glColor3f(1.0, 0.0, 0.0);
   glVertex3f(10.0, 10.0, 0.0); // Vertex 1
   glColor3f(0.0, 1.0, 0.0);
   glVertex3f(70.0, 30.0, 0.0); // Vertex 2
   ...
   glColor3f(0.0, 0.0, 0.0);
   glVertex3f(30.0, 30.0, 0.0); // Vertex 8 = Vertex 0
   glColor3f(1.0, 0.0, 0.0);
   glVertex3f(10.0, 10.0, 0.0); // Vertex 9 = Vertex 1
glEnd();
```

squareAnnulus1.cpp

- a plain-vanilla program which draws the square annulus

(a)



(b)

squareAnnulus2.cpp

```
// Vertex co-ordinate vectors.
static float vertices[8][3] =
{
    { 30.0, 30.0, 0.0 },
    { 10.0, 10.0, 0.0 },
    { 70.0, 30.0, 0.0 },
    { 90.0, 10.0, 0.0 },
    { 70.0, 70.0, 0.0 },
    { 90.0, 90.0, 0.0 },
    { 30.0, 70.0, 0.0 },
    { 10.0, 90.0, 0.0 }
};

// Vertex color vectors.
static float colors[8][3] =
{
    { 0.0, 0.0, 0.0 },
    { 1.0, 0.0, 0.0 },
    { 0.0, 1.0, 0.0 },
    { 0.0, 0.0, 1.0 },
    { 1.0, 1.0, 0.0 },
    { 1.0, 0.0, 1.0 },
    { 0.0, 1.0, 1.0 },
    { 1.0, 0.0, 0.0 }
};
```

```
glBegin(GL_TRIANGLE_STRIP);
   for(int i = 0; i < 10; ++i)
   {
      glColor3fv(colors[i%8]);
      glVertex3fv(vertices[i%8]);
   }
glEnd();
```

squareAnnulus3.cpp

- Vertex array data structures provided by OpenGL

```
// Draw square annulus.
glBegin(GL_TRIANGLE_STRIP);
// The i th vertex in vertices[] and i th color in colors[]
// are called together by glArrayElement(i).
for (int i = 0; i < 10; ++i) glArrayElement(i % 8);
glEnd();
```

squareAnnulus3.cpp

- Vertex array data structures provided by OpenGL
- Enabled by calling glEnableClientState(array)
- The data is specified with a call to
  - glVertexPointer(size , type , stride , *pointer)
  - glColorPointer(size , type , stride , *pointer)

```
void setup(void)
{
    glClearColor(1.0, 1.0, 1.0, 0.0);

    // Enable two vertex arrays: co-ordinates and color.
    glEnableClientState(GL_VERTEX_ARRAY);
    glEnableClientState(GL_COLOR_ARRAY);

    // Specify locations for the co-ordinates and color arrays.
    glVertexPointer(3, GL_FLOAT, 0, vertices);
    glColorPointer(3, GL_FLOAT, 0, colors);
}
```

squareAnnulus4.cpp

```
// Draw square annulus.
glBegin(GL_TRIANGLE_STRIP);
// The i th vertex in vertices[] and i th color in colors[]
// are called together by glArrayElement(i).
for (int i = 0; i < 10; ++i) glArrayElement(i % 8);
glEnd();
```

```
// Triangle strip vertex indices in order.
static unsigned int stripIndices[] = { 0, 1, 2, 3, 4, 5, 6, 7, 0, 1 };
```

```
// Draw square annulus. glDrawElements() "pulls up"
// data for 10 vertices in one command -
// more efficient than calling glArrayElement() 10 times.
glDrawElements(GL_TRIANGLE_STRIP, 10, GL_UNSIGNED_INT, stripIndices);
```

squareAnnulus4.cpp

- glDrawElements(*primitive, countIndices, type, *indices*)
  - *primitive* is a geometric primitive,
  - *indices* is the address of the start of an array of indices,
  - *type* is the data type of the indices array
  - *countIndices* is the number of indices to use.

glDrawElements (*primitive*, *countIndices*, *type*, **indices*) ;

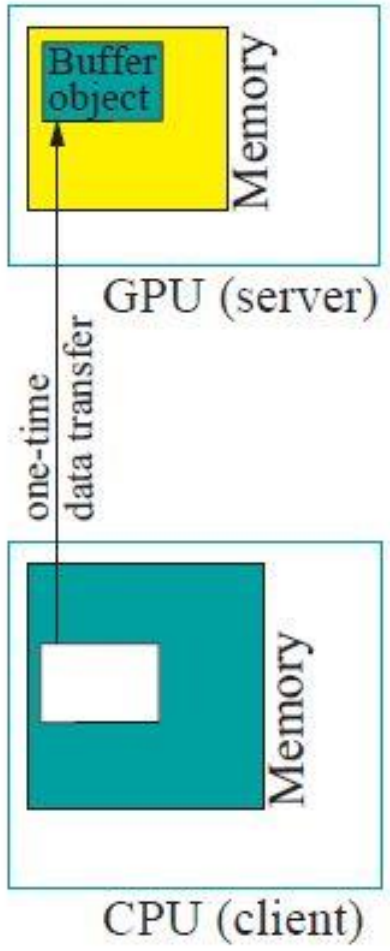
```
glBegin(primitive);
   for(i = 0; i < countIndices; i++) glArrayElement(indices[i]);
glEnd();
```

Immediate mode v.s. Retained mode

- Immediate mode rendering
  - glBegin()-glEnd() -type drawing commands
  - the client (the machine running the program) forces rendering by the server (the GPU)
- Retained mode rendering
  - glDrawElements() and a few more of their relatives
  - the client provides the server only with instructions to perform and the data to use, allowing the server to optimize prior to rendering
- Drawing calls to the GPU is the fewer the better because of per-call initialization cost

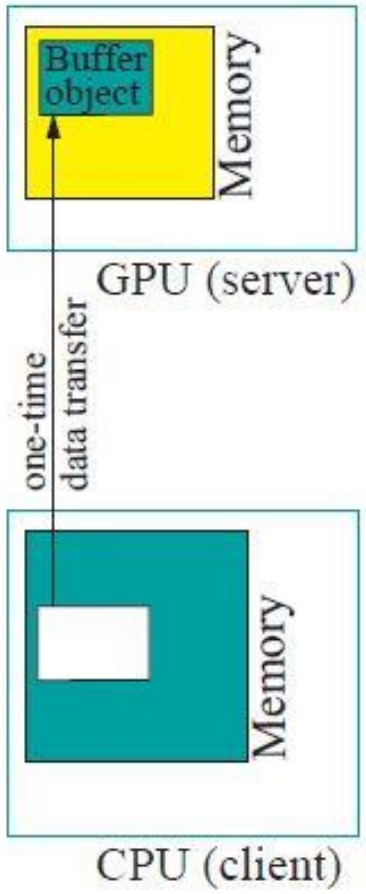
Vertex Buffer Objects

- OpenGL's client-server model
  - glDrawElements() – tranfer data from the CPU to the GPU
  - Accessing data across a bus is much slower than accessing it locally
  - The access might even be redundant if the same data had been retrieved for an earlier command and, subsequently, not changed.
- Vertex Buffer Objects
  - Allow the programmer to explicitly ask that some particular set of data be shipped from the client to the server and stored there for future use



Vertex Buffer Objects

```
// Initialization routine.
void setup(void)
{
     glGenBuffers(2, buffer); // Generate buffer ids.
     // Bind vertex buffer and reserve space.
     glBindBuffer(GL_ARRAY_BUFFER, buffer[VERTICES]);
     glBufferData(GL_ARRAY_BUFFER, sizeof(vertices) +
          sizeof(colors), NULL, GL_STATIC_DRAW);
     // Copy vertex coordinates data into first half of vertex buffer.
     glBufferSubData(GL_ARRAY_BUFFER, 0, sizeof(vertices), vertices);
     // Copy vertex color data into second half of vertex buffer.
     glBufferSubData(GL_ARRAY_BUFFER, sizeof(vertices),
          sizeof(colors), colors);
     // Bind and fill indices buffer.
     glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, buffer[INDICES]);
     glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(stripIndices),
          stripIndices, GL_STATIC_DRAW);
     // Enable two vertex arrays: co-ordinates and color.
     glEnableClientState(GL_VERTEX_ARRAY);
     glEnableClientState(GL_COLOR_ARRAY);
     // Specify vertex and color pointers to the start of the respective data.
     glVertexPointer(3, GL_FLOAT, 0, 0);
     glColorPointer(3, GL_FLOAT, 0, (void *)(sizeof(vertices)));
}
```



Display Lists

- A set of commands invoked repeatedly
- The program simply calls the display list rather than reissue them
- Provide a logical way to encapsulate objects, such as a wheel or robot arm

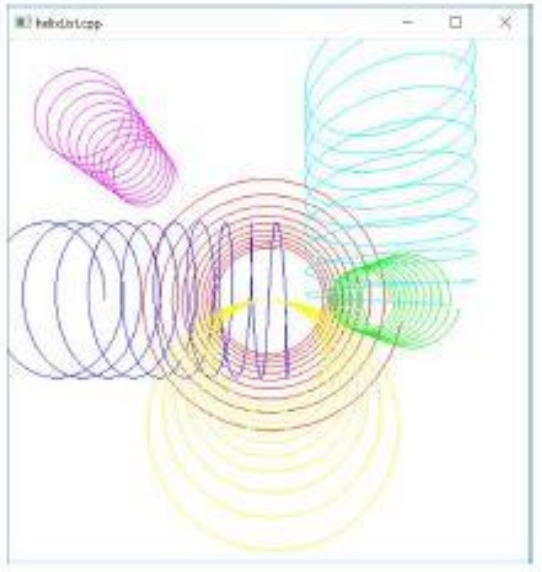
helixList.cpp

```
void setup(void)
{
    float t; // Angle parameter.
    // Return a list index.
    aHelix = glGenLists(1);
    // Begin create a display list.
    glNewList(aHelix, GL_COMPILE);
    // Draw a helix.
    glBegin(GL_LINE_STRIP);
    for (t = -10 * PI; t <= 10 * PI; t += PI / 20.0)
    glVertex3f(20 * cos(t), 20 * sin(t), t);
    glEnd();
    glEndList();
    // End create a display list.
    glClearColor(1.0, 1.0, 1.0, 0.0);
}
```

helixList.cpp

```
// Drawing routine.
void drawScene(void)
{
    glClear(GL_COLOR_BUFFER_BIT);
    glColor3f(1.0, 0.0, 0.0);
    glPushMatrix();
    glTranslatef(0.0, 0.0, -70.0);
    glCallList(aHelix); // Execute display list.
    glPopMatrix();

    glColor3f(0.0, 1.0, 0.0);
    glPushMatrix();
    glTranslatef(30.0, 0.0, -70.0);
    glScalef(0.5, 0.5, 0.5);
    glCallList(aHelix); // Execute display list.
    glPopMatrix();
    GlFlush();
}
```



Drawing Text

- Bitmapped characters
  - glutBitmapCharacter(*font , character)
  - Fonts

```
GLUT_BITMAP_8_BY_13
GLUT_BITMAP_9_BY_15
GLUT_BITMAP_TIMES_ROMAN_10
GLUT_BITMAP_TIMES_ROMAN_24
GLUT_BITMAP_HELVETICA_10
GLUT_BITMAP_HELVETICA_12
GLUT_BITMAP_HELVETICA_18
```

- Stroke characters
  - glutStrokeCharacter(*font , character)
  - Fonts

```
GLUT_STROKE_ROMAN
GLUT_STROKE_MONO_ROMAN
```

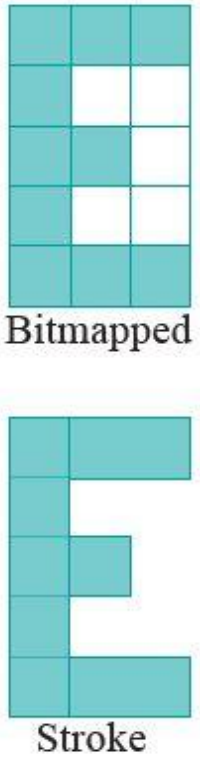


Figure 3.14: Bitmapped versus stroke text.

Drawing Bitmapped Text

```
// Routine to draw a bitmap character string.
void writeBitmapString(void *font, char *string)
{
    char *c;
    for (c = string; *c != '\0'; c++) glutBitmapCharacter(font,
    *c);
}
```

```
glRasterPos3f(10.0, 90.0, 0.0);
writeBitmapString(GLUT_BITMAP_8_BY_13, "GLUT_BITMAP_8_BY_13");
```

Drawing Stroke Text

```
// Routine to draw a stroke character string.
void writeStrokeString(void *font, char *string)
{
    char *c;
    for (c = string; *c != '\0'; c++) glutStrokeCharacter(font, *c);
}
```

```
glTranslatef(40.0, 40.0, 0.0);
glRotatef(-30.0, 0.0, 0.0, 1.0);
glScalef(0.025, 0.025, 0.025);
writeStrokeString(GLUT_STROKE_MONO_ROMAN, "GLUT_STROKE_MONO_ROMAN");
```

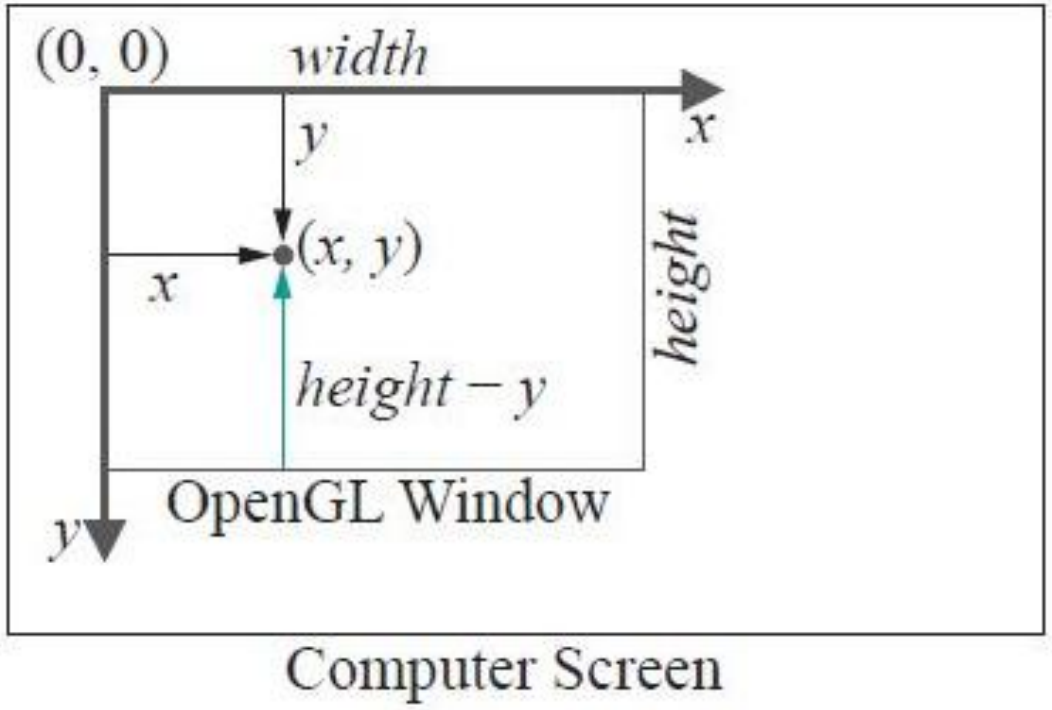
Programming the Mouse

- Register mouse callback routine:
  - glutMouseFunc(*mouse_callback_func*)
  - *mouse_callback_func*(*button, state, x, y*)

  button :
  GLUT_LEFT_ BUTTON, GLUT _RIGHT _BUTTON, GLUT _MIDDLE _BUTTON
  state :
  GLUT_UP, GLUT_DOWN
  x, y :
  location in the *OpenGL window* at which the mouse event occurs

OpenGL Window

- The origin is at the upper-left corner of the OpenGL window
- Units are still pixels
- X–axis heads right and the Y-axis heads down



Mouse Motion

- Register mouse motion callback routine:
  - glutMotionFunc(mouseMotion)
  - mouseMotion(int x, int y) is called be called when the moves *with the button pressed*

- Example: mouseMotion.cpp
- Allows the user to drag the newly created point using the mouse with the left button still pressed.

Turning the Wheel

- Register mouse wheel callback routine:
  - glutMouseWheelFunc(mouseWheel)
  - void mouseWheel(int *wheel*, int *direction*, int *x*, int *y*)
    wheel: the wheel number, which is 0 if there is a single wheel
    direction: the direction of rotation, which is either +1 or − 1
    (*x,y*) : the location of the mouse in window coordinates

- Example: mouseWheel.cpp
- change the size of the last point drawn by turning the mouse wheel

Programming Non-ASCII Keys

- glutKeyboardFunc(keyInput);
- keyInput(unsigned char *key*, int *x*, int *y*)
- ASCII code

USASCII code chart

Bits $b_7 b_6 b_5$ →	0 0 0	0 0 1	0 1 0	0 1 1	1 0 0	1 0 1	1 1 0	1 1 1
b_4 b_3 b_2 b_1 ↓ Row / Column →	0	1	2	3	4	5	6	7
0 0 0 0	0	NUL	DLE	SP	0	@	P	`	p
0 0 0 1	1	SOH	DC1	!	1	A	Q	a	q
0 0 1 0	2	STX	DC2	"	2	B	R	b	r
0 0 1 1	3	ETX	DC3	#	3	C	S	c	s
0 1 0 0	4	EOT	DC4	$	4	D	T	d	t
0 1 0 1	5	ENQ	NAK	%	5	E	U	e	u
0 1 1 0	6	ACK	SYN	&	6	F	V	f	v
0 1 1 1	7	BEL	ETB	'	7	G	W	g	w
1 0 0 0	8	BS	CAN	(	8	H	X	h	x
1 0 0 1	9	HT	EM	)	9	I	Y	i	y
1 0 1 0	10	LF	SUB	*	:	J	Z	j	z
1 0 1 1	11	VT	ESC	+	;	K	[	k	{
1 1 0 0	12	FF	FS	,	<	L	\	l	|
1 1 0 1	13	CR	GS	-	=	M	]	m	}
1 1 1 0	14	SO	RS	.	>	N	^	n	~
1 1 1 1	15	SI	US	/	?	O	_	o	DEL

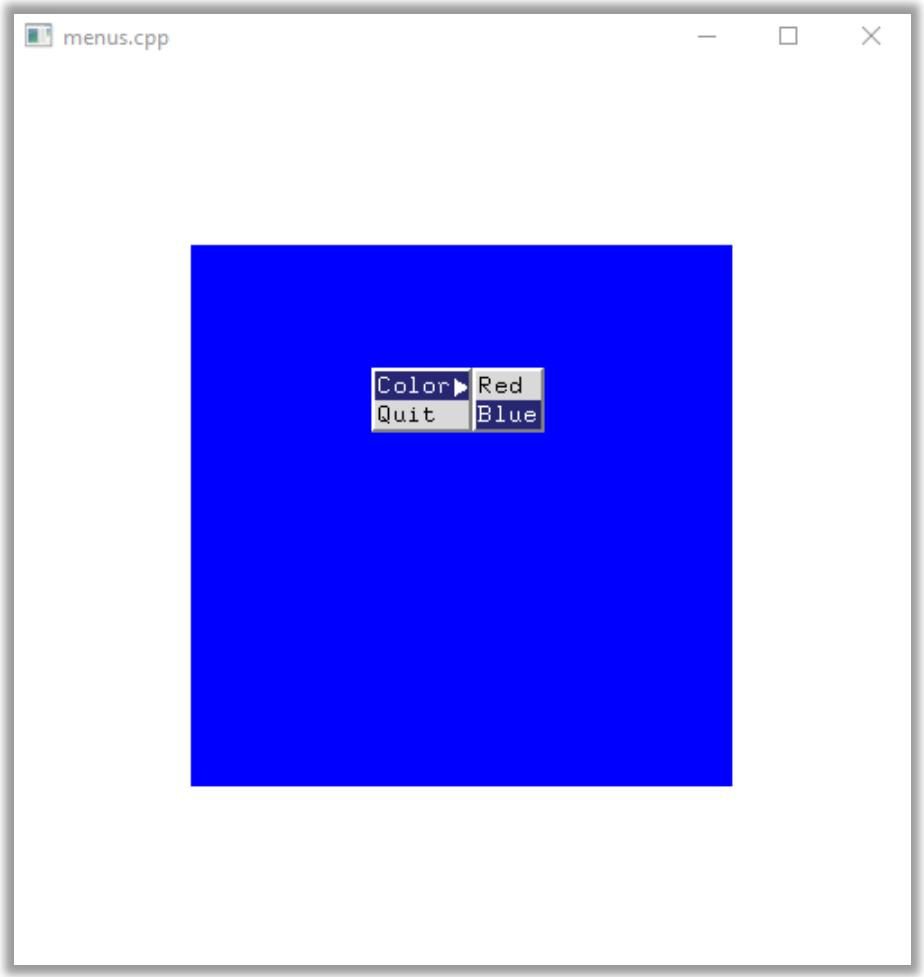
Programming Non-ASCII Keys

- Register Non-ASCII Keys callback routine:
  - glutSpecialFunc(specialKeyInput)
  - specialKeyInput(int *key*, int *x*, int *y*)
    key: a GLUT_KEY_* constant for the special key pressed.
    (*x, y*): location of the mouse in window relative coordinates when the key was pressed.
  - GLUT_KEY_*:
    GLUT_KEY_F1, GLUT_KEY_F2,…, GLUT_KEY_F12
    GLUT_KEY_LEFT, GLUT_KEY_UP, GLUT_KEY_RIGHT, GLUT_KEY_DOWN
    GLUT_KEY_PAGE_UP, GLUT_KEY_PAGE_DOWN, GLUT_KEY_HOME,
    GLUT_KEY_END, GLUT_KEY_INSERT

- Example: \Chapter2\ moveSphere.cpp

Menus

- Example: \Chapter3\menus.cpp

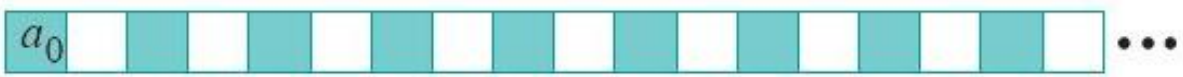


Menus

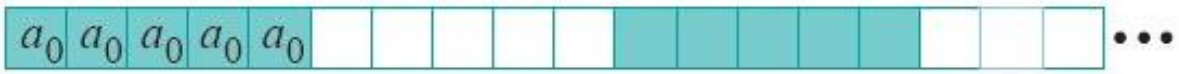
- glutCreateMenu(*menu_function*): creates a menu and registers *menu_function*() as its callback function,
- glutAddMenuEntry(*tag* , returned value): creates a menu item labeled *tag* which, when clicked, returns returned value to the callback function *menu_function*() of the menu itself.
- glutAddSubMenu(tag , sub menu): when tag is clicked a sub-menu pops up whose id is sub menu.
- glutAttachMenu(button): attaches the menu to a mouse button.

Line Stipples

- glEnable(GL_LINE_STIPPLE) : enable stippling
- glDisable(GL LINE STIPPLE) : disable stippling
- glLineStipple(*factor*, *pattern*): specify stipple patterns
  - *pattern:* Specifies a 16-bit integer whose bit pattern determines which fragments of a line will be drawn when the line is rasterized
  - *factor:* Specifies a multiplier for each bit in the line stipple pattern



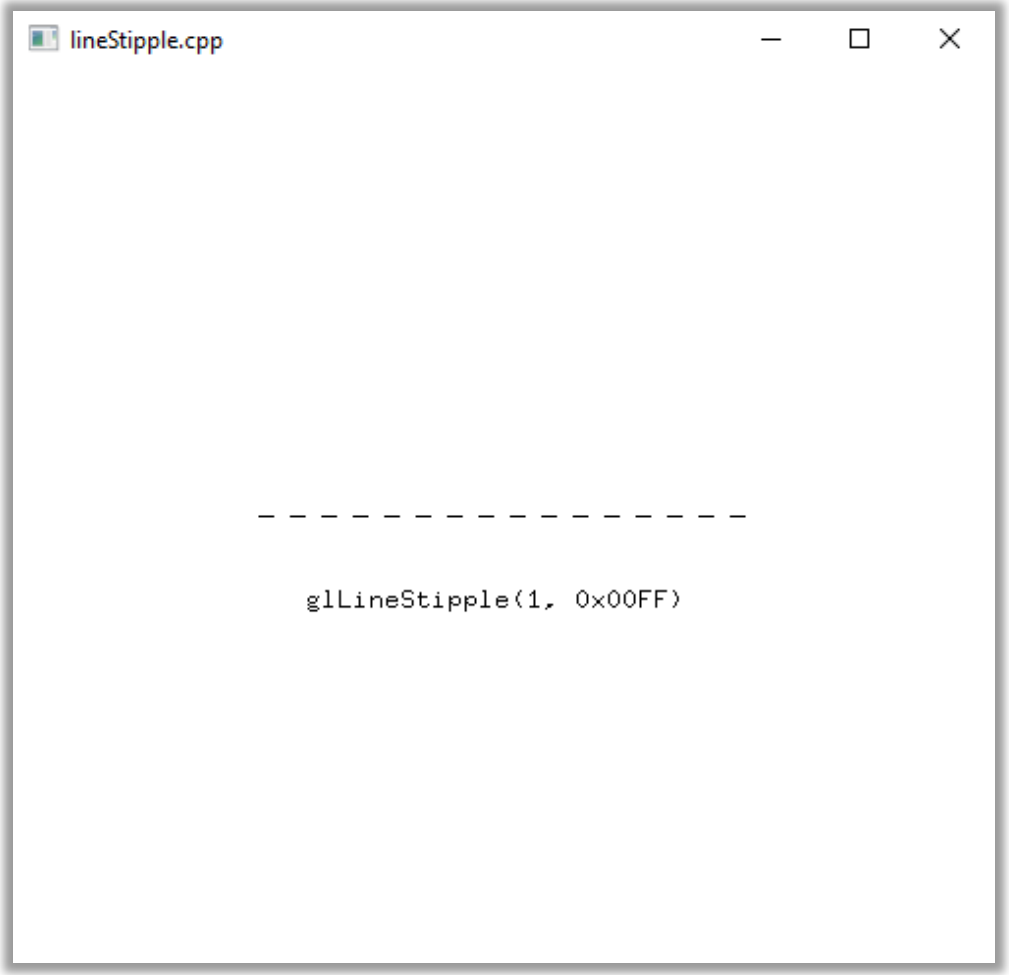
glLineStipple(1, 0x5555) (a) 0x5555=0101010101010101



glLineStipple(5, 0x5555) (b)

Line Stipples

- Example: Chapter3\lineStipple.cpp



GLUT Objects

- The FreeGLUT library offers a collection of standard objects:

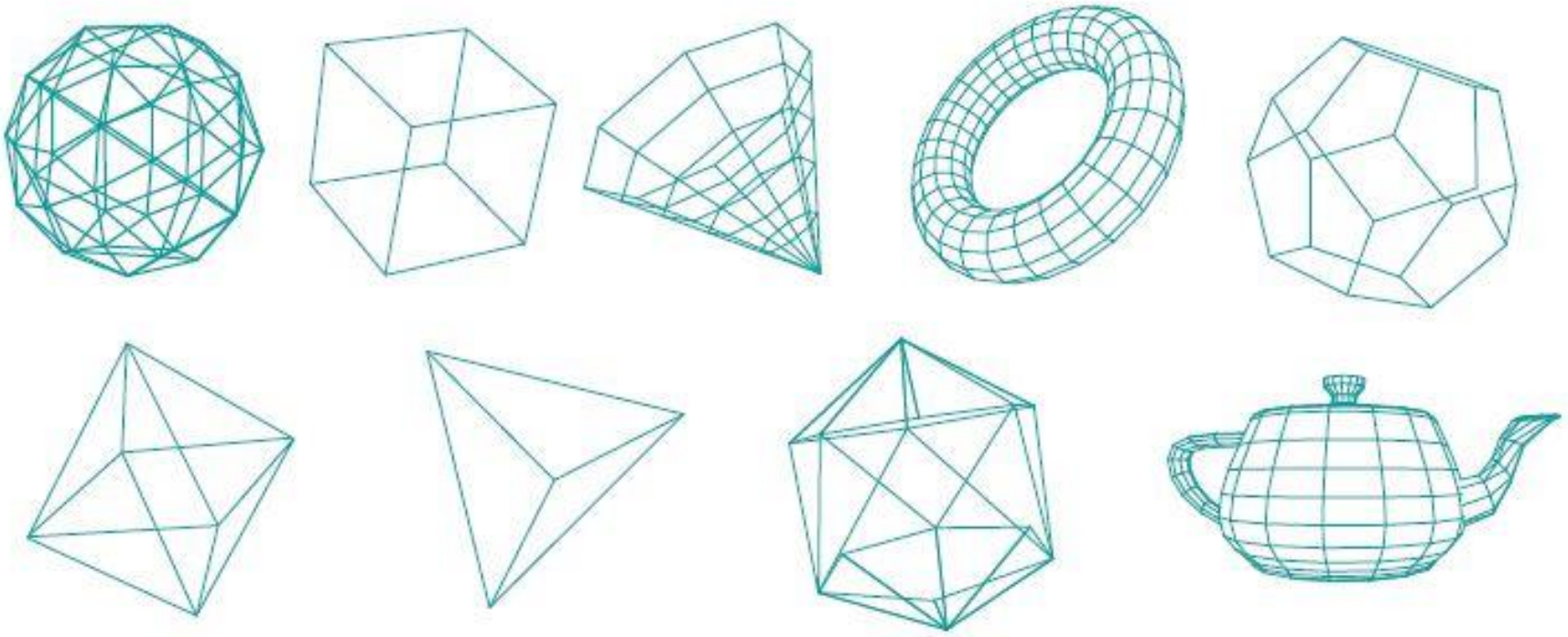


Figure 3.23: Wireframe FreeGLUT objects.

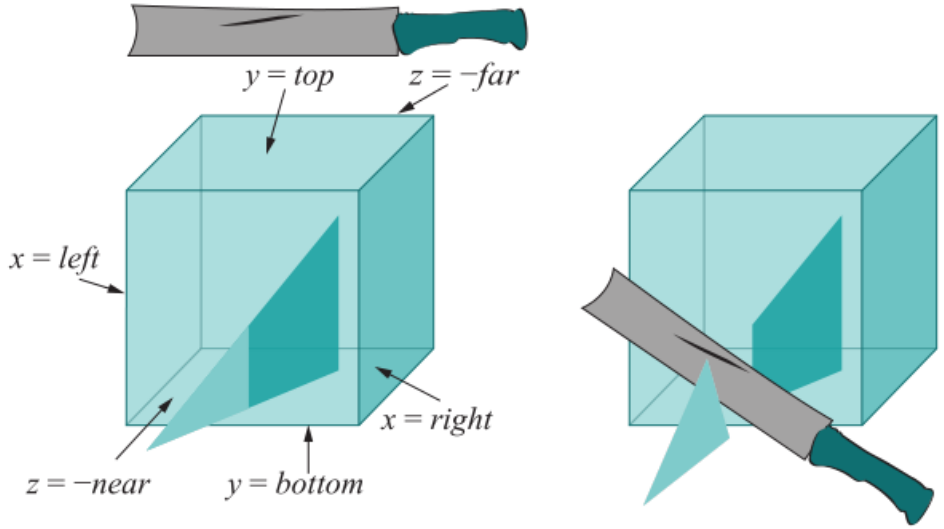
GLUT Objects

- The respective calls are shown in the table below
- The objects are all drawn centered at the origin
- The parameters determine the object's size and the fineness of its mesh

Solid	Wireframe
`glutSolidSphere(`*radius*, *slices*, *stacks*`)`	`glutWireSphere(`*radius*, *slices*, *stacks*`)`
`glutSolidCube(`*size*`)`	`glutWireCube(`*size*`)`
`glutSolidCone(`*base*, *height*, *slices*, *stacks*`)`	`glutWireCone(`*base*, *height*, *slices*, *stacks*`)`
`glutSolidTorus(`*inRadius*, *outRadius*, *sides*, *rings*`)`	`glutWireTorus(`*inRadius*, *outRadius*, *sides*, *rings*`)`
`glutSolidDodecahedron(`*void*`)`	`glutWireDodecahedron(`*void*`)`
`glutSolidOctahedron(`*void*`)`	`glutWireOctahedron(`*void*`)`
`glutSolidTetrahedron(`*void*`)`	`glutWireTetrahedron(`*void*`)`
`glutSolidIcosahedron(`*void*`)`	`glutWireIcosahedron(`*void*`)`
`glutSolidTeapot(`*size*`)`	`glutWireTeapot(`*size*`)`

Clipping Planes

- Six clipping planes of the viewing volume



- User-defined clipping planes:
  - glClipPlane(GL_CLIP_PLANEi, *equation);

Clipping Planes

- User-defined clipping planes:
  - glClipPlane(GL_CLIP_PLANE*i*, **equation*);

    specifies an *i*th additional clipping plane, where *equation* points to an array {A,B,C,D} specifying the coefficients of the equation of the new clipping plane

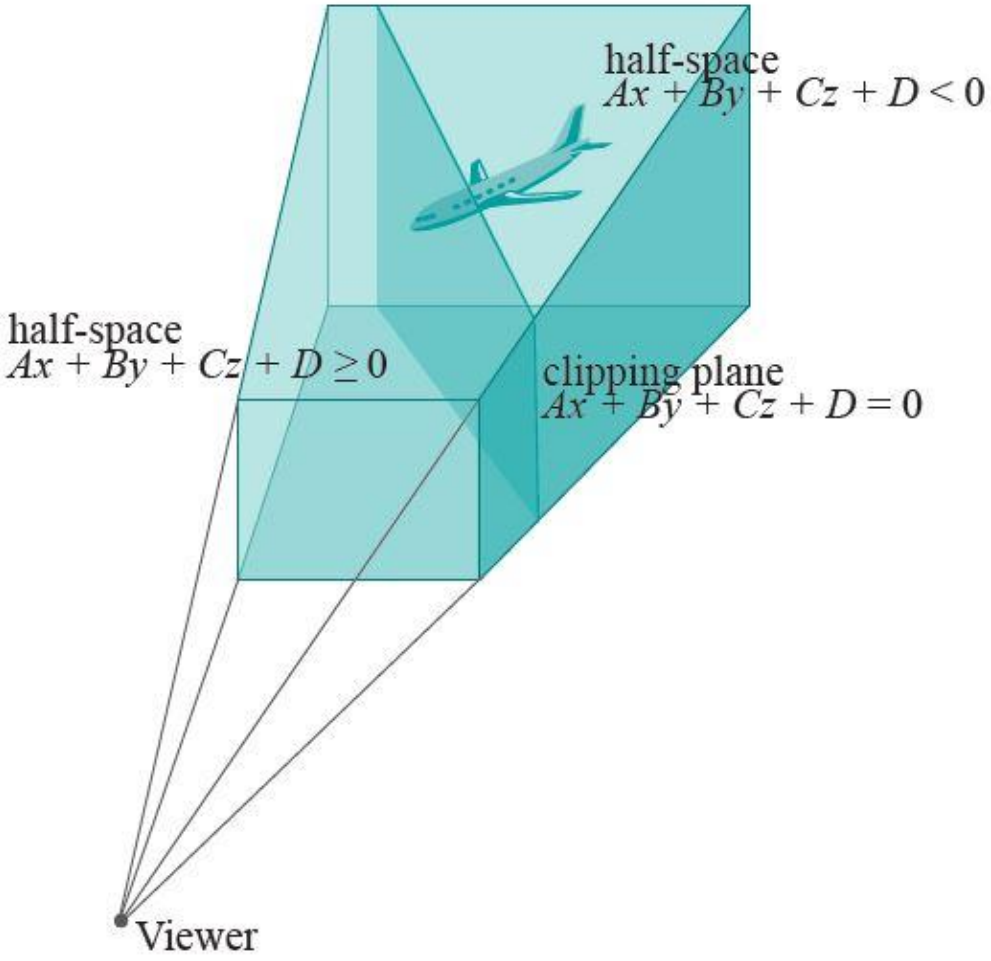
$$Ax + By + Cz + D = 0$$

  - glEnable(GL_CLIP_PLANE*i*)

    points (x,y,z) of objects which lie in the following open half-space are clipped off

$$Ax + By + Cz + D < 0$$

Clipping Planes

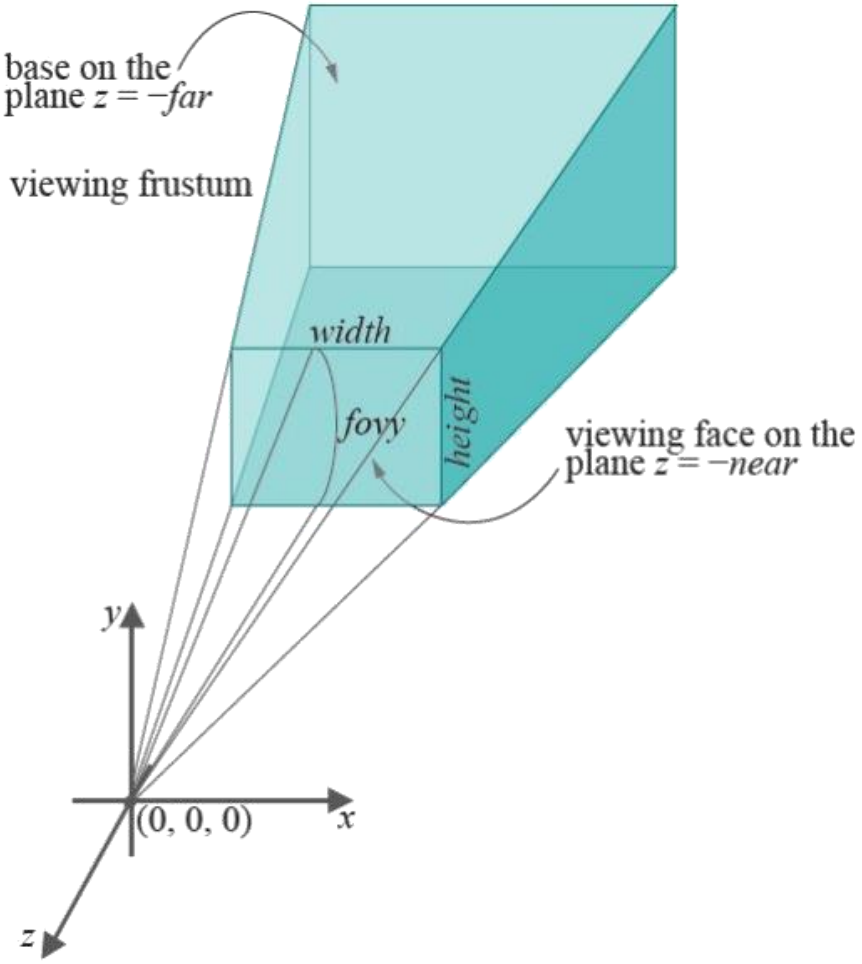


Clipping Planes

- Example: glutObjects.cpp

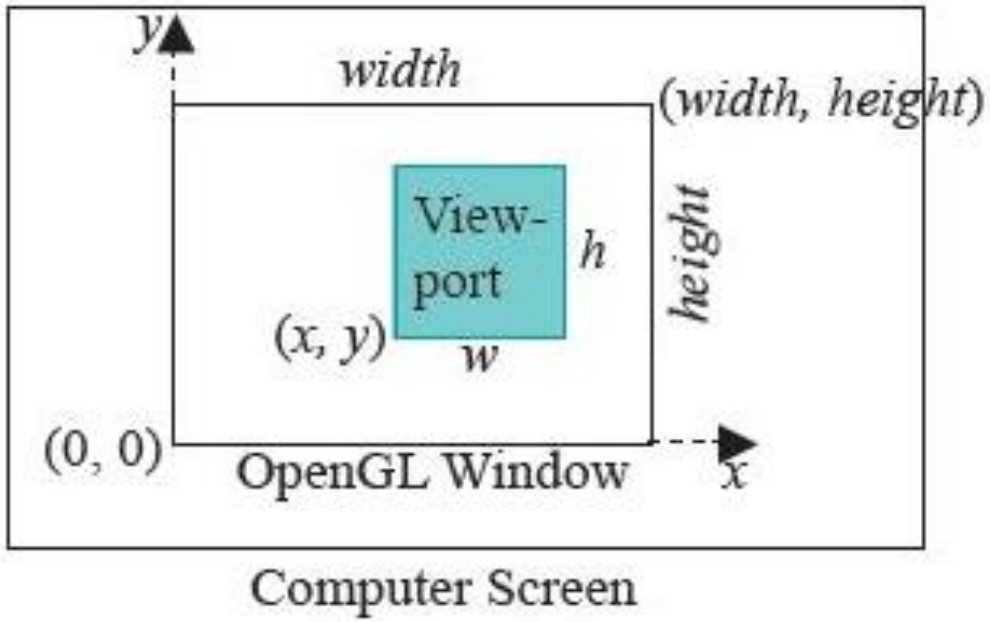
Viewing Frustum- gluPerspective()

- gluPerspective(fovy, aspect, near, far)
  - fovy: the field of view angle,
  - aspect: the aspect ratio = width/height of the front face of the frustum;
  - near:specifies the distance from the viewer to the near clipping plane (always positive).
  - far: specifies the distance from the viewer to the far clipping plane (always positive).



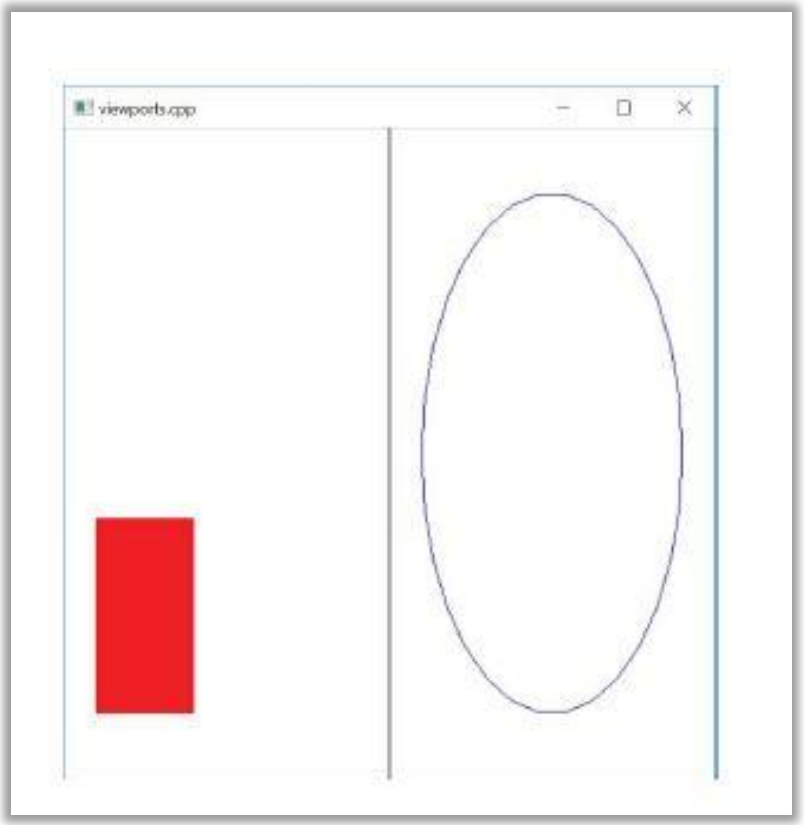
Viewports

- glViewport(x , y , w , h)
  - (x, y): position of the lower-left corner of the viewport,
    the origin is located at the lower-left corner of the OpenGL Window.
    the increasing direction of the x -axis is rightwards,
    the increasing direction of the y-axis is upwards
  - (w, h): the width and height of the viewport, respectively.



Viewports

- Example: viewports.cpp



Multiple Windows

```
// First top-level window specs.
  glutInitWindowSize(250, 500);
  glutInitWindowPosition(100, 100);
  // Create the first window and return id.
  glutCreateWindow("window 1");
  // Initialization, display, resize and keyboard routines of the first window.
  setup1();
  glutDisplayFunc(drawScene1);
  glutReshapeFunc(resize1);
  glutKeyboardFunc(keyInput); // Routine is shared by both windows.
  // Second top-level window specs.
  glutInitWindowSize(250, 500);
  glutInitWindowPosition(400, 100);
  // Create the second window and return id.
  glutCreateWindow("window 2");
  // Initialization, display, resize and keyboard routines of the second window.
  setup2();
  glutDisplayFunc(drawScene2);
  glutReshapeFunc(resize2);
  glutKeyboardFunc(keyInput); // Routine is shared by both windows.
```

Multiple Windows

- Example: multipleWindows.cpp