

|  |  |
| --- | --- |
| **ASSIGNMENT** | |
| **Subject Code** | CSC309A |
| **Subject Name** | Computer Graphics |
| **Programme/Course** | B-Tech |
| **Department** | CSE |
| **Faculty** | FET |

#### 

|  |  |
| --- | --- |
| **Name of the Student** | Akash Bhadauria |
| **Reg. No** | 16ETCS002004 |
| **Semester/Year** | 6TH Sem / 3RD Year |
| **Subject Leader/s** | Mr. Deepak V. |

**Ramaiah University of Applied Sciences**

**University House, Gnanagangothri Campus, New BEL Road,**

**M S R Nagar, Bangalore, Karnataka, INDIA - 560 054.**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Declaration Sheet** | | | | | | | | |
| Student Name | Akash Bhadauria | | | | | | | |
| Reg. No | 16ETCS002004 | | | | | | | |
| Programme/Course | B.Tech | | | | | Semester/Year | 6th Sem / 3rd Year | |
| Subject Code | CSC309A | | | | | | | |
| Subject Title | Computer Graphics | | | | | | | |
| Subject Date |  | | to | |  | | | |
| Subject Leader | Mr. Deepak V | | | | | | | |
| **Declaration**  The assignment submitted herewith is a result of my own investigations and that I have conformed to the guidelines against plagiarism as laid out in the Student Handbook. All sections of the text and results, which have been obtained from other sources, are fully referenced. I understand that cheating and plagiarism constitute a breach of University regulations and will be dealt with accordingly. | | | | | | | | |
| Signature of the Student | |  | | | | | Date |  |
| Submission date stamp  (by Examination & Assessment Section) | |  | | | | | | |
| Signature of the Subject Leader and date | | | | Signature of the Reviewer and date | | | | |
|  | | | |  | | | | |

# **Question No. 1**

**Solution to Question No. 1:**

**PART A**

**A.1.1 Introduction**

In the quest for more realistic imagery, one of the most frequent criticisms of early synthesized raster images was the extreme smoothness of surfaces - they showed no texture, bumps, scratches, dirt, or fingerprints. Realism demands complexity, or at least the appearance of complexity. **Texture mapping** is a relatively efficient means to create the appearance of complexity without the tedium of modelling and rendering every 3-D detail of a surface. The study of texture mapping is valuable because its methods are applicable throughout computer graphics and image processing. The technique has been applied to a number of surface attributes: surface colour, surface normal, specularity, transparency, illumination, and surface displacement, to name a few. (Graham, 2003)

**A.1.2 Description of techniques to minimize the artifacts.**

* Tedious to specify texture coordinates.
* Acquiring textures is surprisingly difficult:

1. Photographs have projective distortions.
2. Variation in reflectance and illumination.

* The texture and object are in two different spaces.
* Where in the rendering pipeline do we specify this mapping?

1. Object or world space?
2. Map onto untransformed surfaces

* Texture filtering: A point on the surface maps to a location between texels in the texture.
* **Problem with point sampling texel colour:** A **texel** is a smallest square area in a texture mapping image. When a large texture area is mapped to a small image area, a large number of texels will be mapped to a single pixel. If you point-sample, i.e., if use the colour of a single texel to paint the pixel, aliasing happens and the resulting texture mapping won't look nice. To avoid this aliasing problem, texture needs to be filtered (i.e., the colours of texels need to be averaged.)
* **Problem with averaging texel colours:** For instance, in a perspective view of a plane, a single pixel in the final image may correspond to thousands of texels in the source image. Determining the final colour for a single pixel would require a lot of computational time if the renderer had to average the colours of all the corresponding texels. Hence, averaging a number of the texel colours for each pixel is a bad idea because it will take a lot of rendering time.

**Minimizing the artifacts:**

1. **Mip-mapping** is a method that prevents the aliasing problem caused by point sampling and the costly rendering time by averaging texel colours for each pixel. Mip-mapping pre-filters a texture by creating multiple copies of a texture mapping image, all derived by averaging down the original image to successively lower resolutions. Each image in the sequence is at exactly half the resolution of the previous. The renderer prepares prefiltered "mip-mapped" textures before starts rendering. During the rendering process it chooses the appropriate level of texture image to use, depending on how much screen coverage there is and how obliquely the object is being viewed. The further away or more obliquely, the more the renderer tends to use a lower-resolution (i.e., blurry) version of the texture. (Heckbert, 2009)
2. **Filtering:**

* Take the average of multiple texels to obtain the final RGB value •
* Typically used along with mipmapping.
* Bilinear filtering

– Average the four surrounding texels

– Cheap, and eliminates some aliasing, but does not help with visible LOD divisions (demonstration movies).

* Trilinear filtering

– Interpolate between two LODs

– Final RGB value is between the result of a bilinear filter at one LOD and a second bilinear filter at the next LOD

– Eliminates “seams” between LODs

– At least twice as expensive as bilinear filtering.

* Anisotropic Filtering

– Basic filtering methods assume that a pixel on-screen maps to a square (isotropic) region of the texture

– For surfaces tilted away from the viewer, this is not the case!

– A pixel may map to a rectangular or trapezoidal section of texels—shape filters accordingly and use either bilinear or trilinear filtering.

– Complicated, but produces very nice results.

**A.1.3 Conclusion**

Texture mapping has become a widely used technique because of its generality and efficiency. It has even made its way into everyday broadcast TV, thanks to new real-time video texture mapping hardware such as the Ampex ADO and Quantel Mirage. Rendering systems of the near future will allow any conceivable surface parameter to be texture mapped. Although texture maps are usually much more compact than brute force 3-D modelling of surface details, they can be bulky, especially when they represent a high resolution image as opposed to a low resolution texture pattern that is replicated numerous times. Keeping several of these in random access memory is often a burden on the rendering program. This problem is especially acute for rendering algorithms that generate the image in scanline order rather than object order, since a given scanline could access hundreds of texture maps. (Huang, 2004)

# **Question No. 2**

**Solution to Question No. 2:**

**PART B**

**B1**

**B1.1 Introduction**

After understanding the basics of drawing shapes like triangles and rectangles in Assignment 1, here we take another step and try to translate, rotate, scale the triangle and display the results on the screen. These operations are called transformations (affine transformations).

**Translation:**

One can simply add a translation distance for each direction (x and y) to each component of the coordinates and translate it. The goal is to translate the point p(x,y) to the point p’(x’,y’), so the translation distance for x and y direction is Tx and Ty respectively.

**Equation 1.1**

x’ = x + Tx

y’ = y + Ty

These simple equations can be implemented in an OpenGL program just by adding command:

**glTranslatef(Tx, Ty,Tz)**

**Rotation:**

Rotation is a little more complex than translation because one has to specify multiple items of information. The following three items are required:

* Rotation axis (the axis the shape will be rotated around)
* Rotation direction (the direction: clockwise or counter clockwise)
* Rotation angle (the number of degrees the shape will be rotated through)

**Equation 2.2 (Expression to calculate the rotation)**

x’ = x cos β – y sin β

y’ = x sin β + y cos β

These simple equations can be implemented in an OpenGL program just by adding command:

glRotatef(A,x,y,z) where A is the angle of rotation.

**Scaling:**

Transformation matrix for scaling using the same assumption that the original point is p and the point after scaling is p’. Assuming the scaling factors for the x-axis and y-axis are Sx and Sy respectively, following equations are obtained:

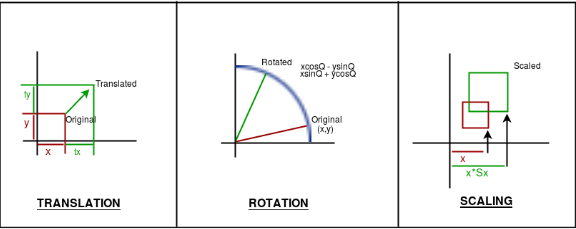
**Equation 3.3**

x’ = Sx × x

y’ = Sy × y

These simple equations can be implemented in an OpenGL program just by adding command:

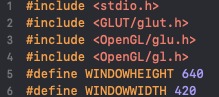
**glScalef(floatx, floaty, floatz)**

****

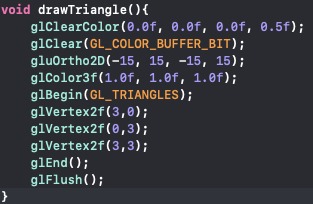
**Fig1.1: Transformation in Computer Graphics**

**Source: https://cdncontribute.geeksforgeeks.org/wp-content/uploads/transformation-computer-graphics.png**

**B2.2 Implementation of Transformation:**

****

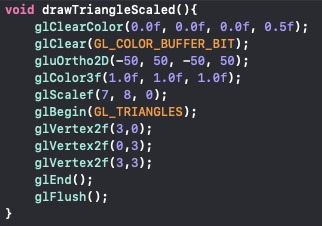
**Fig 1.2: Required Header files**

****

**Fig1.3: Code for drawing a triangle**

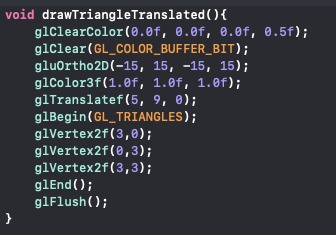
In Fig 1.3, code to draw a triangle of vertex (3,0) (0,3) (3,3) is displayed.

* **glClearColor** (GLclampf *red*, GLclampf *green*, GLclampf *blue*, GLclampf *alpha*) — glClearColor specifies the red, green, blue, and alpha values used by glClear to clear the color buffers. Values specified by glClearColor are clamped to the range 0 1.
* **glClear** sets the bitplane area of the window to values previously selected by glClearColor, glClearDepthf, and glClearStencil.
* **glBegin():**This function delimits the vertices of a primitive or a group of like primitives
* **gluOrtho2D**(left,right,bottom,top)- sets up 2D orthographic viewing region for the current window in above code its sets the coordinates system from -15 to 15 x- axis and -15 to 15 y-axis.
* **glColor3f():**Set the current color
* **glVertex2f():**Set the vertex position
* **glEnd():**It is to plot the points described with glBegin().
* **glFlush():** Force execute of GL commands in finite time



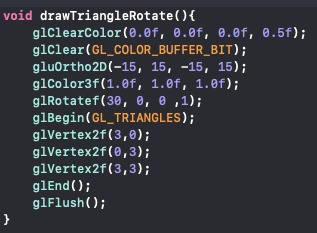
**Fig 1.4 Code to scale the triangle**

In fig 1.4 we scale the triangle by factors of 7 in the x-axis, 8 in the y-axis and 0 in z-axis. These numbers will get multiplied in the current matrix and result will be displayed. The only additional OpenGL command used in this snipped of code is glScalef() which is used to zoom in and out an object in different scales across axes.



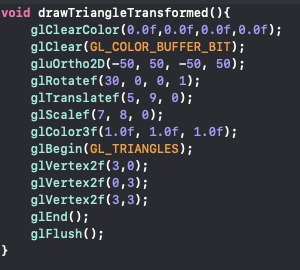
**Fig 1.5 Code to translate the triangle**

In fig 1.5 triangle is moved from its original position by a factor of (5,9,0) to different position on screen. These values will be added to the original coordinates to obtain a different position on screen.The only additional OpenGL command used in this snipped of code is glTranslatef() which is used to set the amount of translation specified.



**Fig 1.6: Code to rotate the triangle**

In fig 1.6 triangle is rotated by 30 degrees along z-axis. The command used for rotation in OpenGL is glRotatef(). The parameter specified in the code are (30,0,0,1) which rotate the triangle by 30 degrees along z-axis, as value of z is set to 1 while x and y is set to 0.



**Fig 1.7 Code for complete transformation of triangle**

In fig 1.7 triangle is first rotated by(30,0,0,1), then translated by (5,9,0) and finally scaled by (7,8,0) to obtain the completely transformed triangle.

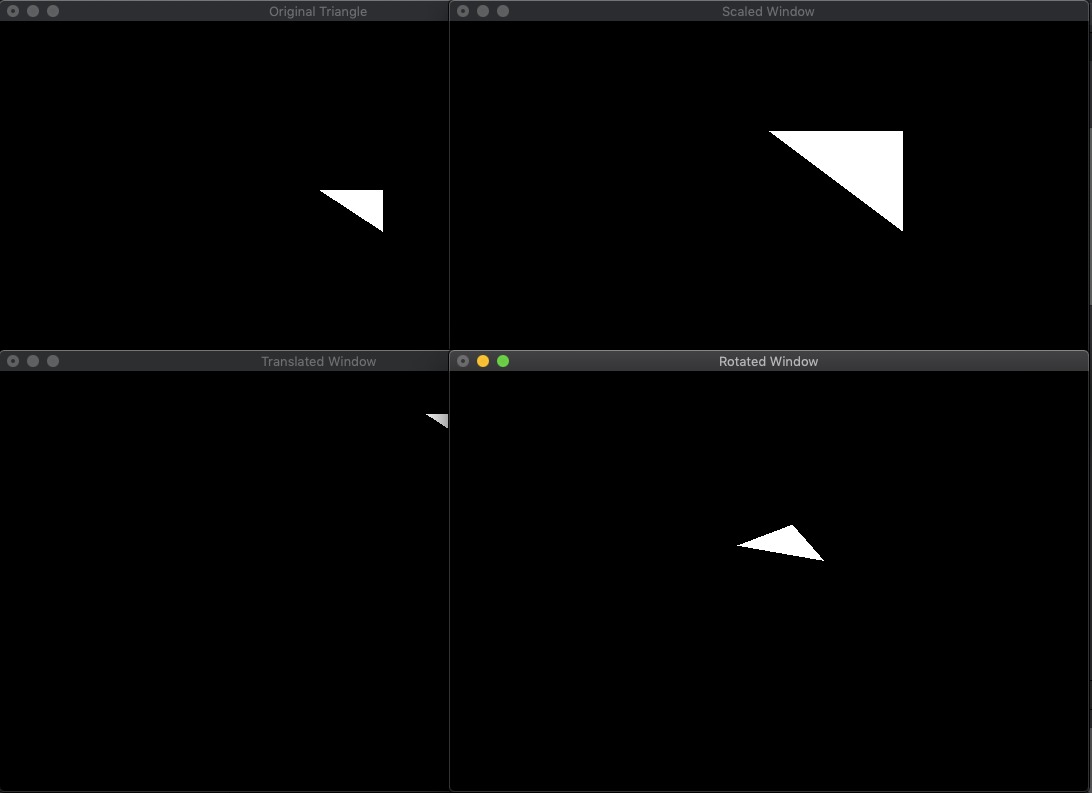
****

**Fig 1.8 Displays the creation of 5 different OpenGL window**

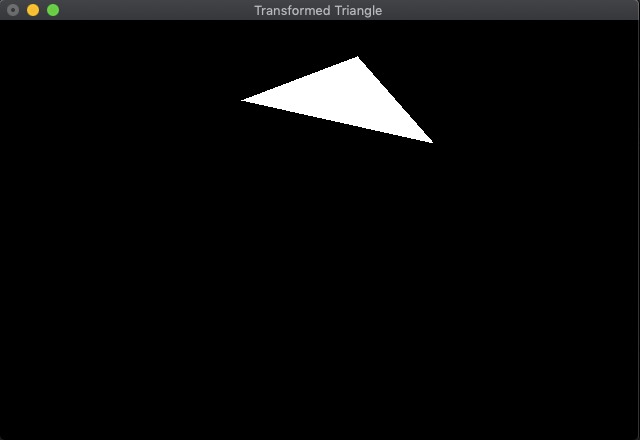
In fig 1.8 5 different OpenGL windows are created and each of them is registered with original triangle, scaled triangle, translated triangle, rotated triangle and transformed triangle.

* **glutInit(&argc,argv):** This function is used to initialize the GLUT library. The pointer to the program’s unmodified argc variable from main. Upon return the value pointed to by argc will be updated, as glutInit extracts any command line options intended for the GLUT library. The program’s unmodified argv variable from main. GlutInit extracts any command line options understood by the GLUT library.
* **glutInitDisplayMode**(unsigned int mode): This function sets the initial display mode. GLUT\_DOUBLE is used to bit mask to select a double buffered window. GLUT\_RGB, bit mask to select an RGBA mode window. GLUT\_DEPTH, bit mask to select a window with a depth buffer.
* **glutInitWindowSize(X,Y):** This function is used to set the size of the window that was created using glutCreateWindow(), the width and height of the window is provided as parameters.
* **glutCreateWindow**(header:String): This function is used to create a window with the heading, which can be NULL or some string. This function is necessary, as the window will be created by this function, and in the window the result would be displayed.
* **glutInitWindowPosition(X,Y**): This function is used to set the position of the window in the display monitor, so that the window is not out of the range.
* **glutDisplayFunc(display):** Sets the display callback for the current window. When GLUT determines that the normal plane for the window needs to be redisplayed, the display callback for the window is called.
* **glutMainLoop():**This function enters the GLUT event processing loop. This routine should be called at most once in a GLUT program. Once called, this routine will never return. It will call as necessary any callbacks that have been registered.

**B.1.3 Results (screenshots and discussion)**



**Fig 1.9 Displays the output for Original, rotated, scaled and translated triangle**



**Fig 1.10 Displays the output for transformed triangle**

# **Question No. 3**

**Solution to Question No. 3:**

**B2.1** **Explanation of the algorithm**

**Main Program Algorithm:**

* Create a window with appropriate size of the window, and position the window appropriately
* CreateviewPort1, viewPort2 and keys[3] of type bool.
* Inscribe display as display function.
* Inscribe keyPressed() as keyboard function

**Display function Algorithm:**

* Viewport is created in lower half of the screen.
* draw() function is called to draw in viewport.
* Then viewport is created in upper half of the screen.
* draw2() is called draw in the second viewport.

**draw() function algorithm:**

* If the viewPort1 is true, then color buffer bit is cleared and changeShape() function is called.

**draw2() function algorithm:**

* If viewPort2 is true, then color buffer bit is cleared and changeShape() function is called.

**changeShape() function algorithm:**

* If keys[0] is true, draw square shape and set keys[0] to false.
* If keys[1] is true, draw triangle shape and set keys[1[ to false.
* If keys[2] is true, draw wired tea pot and set keys[2] to false.

**keyPressed() function algorithm:**

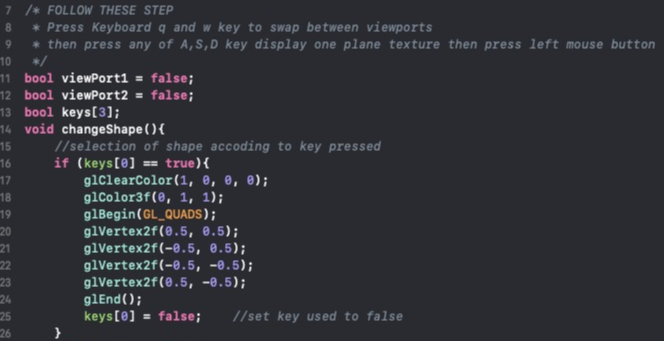
* When key a is pressed, then keys[0] is set to true.
* When key s is pressed, then keys[1] is set to true.
* When key d is pressed, then keys[3] is set to true.
* When left arrow key is pressed on the keyboard, viewPort1 is set to true and viewPort2 is set to false.
* When right arrow key is pressed on the keyboard, viewPort2 is set to true and viewPort1 is set to false.

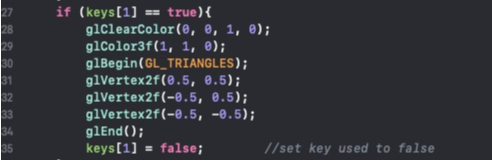
**B2.2 Description of functions:**

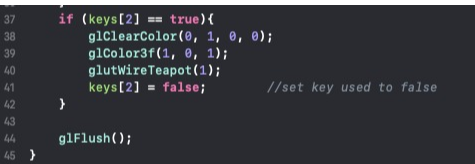
* **glClearColor** (GLclampf *red*, GLclampf *green*, GLclampf *blue*, GLclampf *alpha*) — glClearColor specifies the red, green, blue, and alpha values used by glClear to clear the color buffers. Values specified by glClearColor are clamped to the range 0 1.
* **glClear** sets the bitplane area of the window to values previously selected by glClearColor, glClearDepthf, and glClearStencil.
* **glBegin():**This function delimits the vertices of a primitive or a group of like primitives.
* **glColor3f():**Set the current color
* **glVertex2f():**Set the vertex position
* **glEnd():**It is to plot the points described with glBegin().
* **glFlush():** Force execute of GL commands in finite time
* **GLUT\_KEY\_LEFT** and **GLUT\_KEY\_RIGHT** are used to check whether left or right arrow key was pressed.
* **GL\_QUADS**: Treats each group of four vertices as an independent quadrilateral
* **glViewport(x,y, width, height)** : Sets the viewport
* **glScissor(x,y,width,height)** defines the scissor box meaning texture cannot be drawn outside the box.
* **glEnable():** Enables OpenGL capabilities.
* **glDisable():** Disables OpenGL capabilities.
* **glutInit(&argc,argv):** This function is used to initialize the GLUT library. The pointer to the program’s unmodified argc variable from main. Upon return the value pointed to by argc will be updated, as glutInit extracts any command line options intended for the GLUT library. The program’s unmodified argv variable from main. GlutInit extracts any command line options understood by the GLUT library.
* **glutInitDisplayMode(unsigned int mode):** This function sets the initial display mode. GLUT\_DOUBLE is used to bit mask to select a double buffered window. GLUT\_RGB, bit mask to select an RGBA mode window. GLUT\_DEPTH, bit mask to select a window with a depth buffer.
* **glutInitWindowSize(X,Y):** This function is used to set the size of the window that was created using **glutCreateWindow(),** the width and height of the window is provided as parameters.
* **glutInitWindowPosition(X,Y):** This function is used to set the position of the window in the display monitor, so that the window is not out of the range.
* **glutCreateWindow(header:String):** This function is used to create a window with the heading, which can be NULL or some string. This function is necessary, as the window will be created by this function, and in the window the result would be displayed.
* **glutDisplayFunc(display):** Sets the display callback for the current window. When GLUT determines that the normal plane for the window needs to be redisplayed, the display callback for the window is called.
* **glutMainLoop():**This function enters the GLUT event processing loop. This routine should be called at most once in a GLUT program. Once called, this routine will never return. It will call as necessary any callbacks that have been registered.

****

**Fig2.1 Required header files**

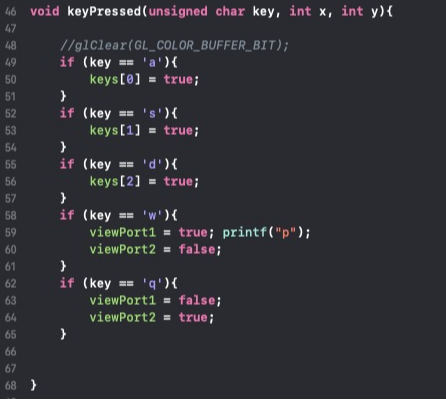
****

****

****

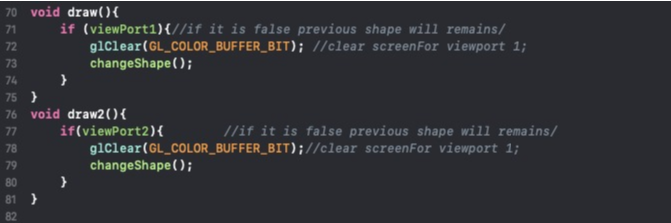
**Fig 2.2 Displays the changeShape()function**

In fig 2.2 variables viewPort1, viewPort2 and keys[3] are defined to check which keys were pressed and for swapping between 2 viewports. When keys[0] is true, it means key ‘a’ is pressed on the keyboard and square shape will be drawn in the selected viewport. The keys[0] is set to false after the shape is being drawn so next time it does not overlap with the next shape. Similarly, when key ‘s’ and ‘d’ are pressed on the keyboard triangle and wired tea pot will be drawn in selected viewport**.**

****

**Fig 2.3 Displays the keyPressed() function**

When user will press left arrow key the viewPort1 will be set to true and viewport2 will be set to false specifying that texture will be drawn in viewport 1 and vice versa. When user will press right arrow key GLUT\_KEY\_LEFT and GLUT\_KEY\_RIGHT are used to check whether left or right arrow key was pressed.



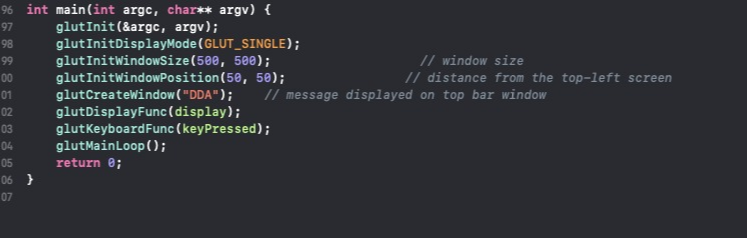
**Fig 2.4 displays the draw() and draw2() functions**

If viewPort1 is selected, then draw the shape in viewPort1 else no update on the screen will be made. Similarly, when viewPort2 is true shape will be drawn in viewPort2 else no update on the screen will be made.

****

**Fig 2.5 code for display() function**

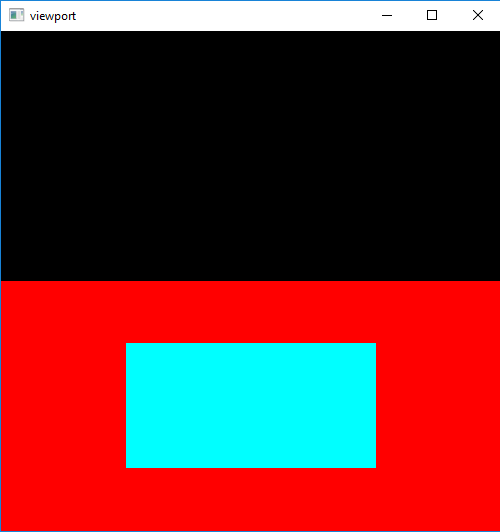
In fig 2.5, 2 viewports are created using **glViewport(x,y,width,height)** where coordinates **x** and **y** specifies the lower left corner of the screen and **width** and **height** specifies width and height respectively. **glScissor(x,y,width,height)** defines the scissor box meaning texture cannot be drawn outside the box, coordinated **x** and **y** specifies the lower left corner of the screen and **width** and **height** specifies width and height respectively. This function is enabled and disabled by using glEnable() and glDisable().



**Fig 2.6: Code for Main Program**

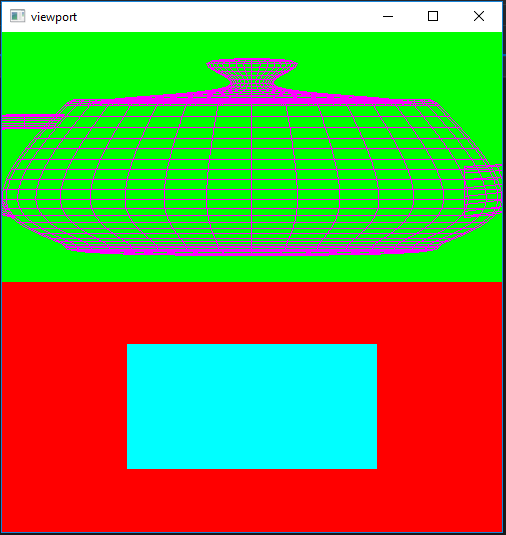
In fig 2.6, Display function for the window is set using glutDisplaufunc() and glutKeyboardFunc() is used to register a function whenever a key is pressed on the keyboard.

**B2.3: Results (screenshots and discussion)**



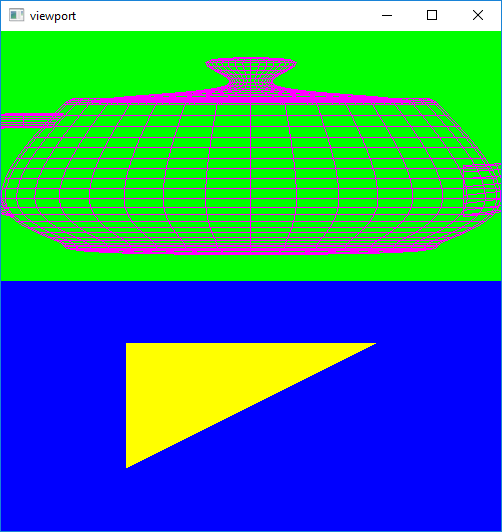
**Fig 2.7 Display the output when square is drawn in selected viewport**

In fig 2.7, when left arrow key was pressed, lower view port got selected and by pressing ‘q’ key a square was drawn in the selected viewport while nothing was drawn in upper viewport.



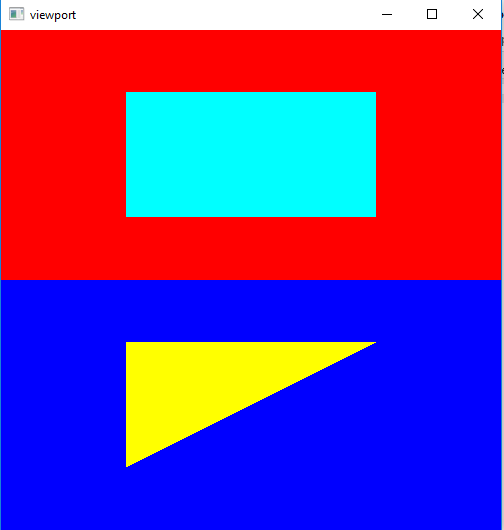
**Fig 2.8 Displays the output when wired teapot is drawn in the selected viewport**

In fig 2.8, when right arrow key was pressed upper view port got selected and then by pressing ‘d’ key wired tea pot was drawn in upper view port leaving lower view port unaffected.



**Fig 2.9 Displays the output when square is replaced via triangle in selected viewport**

In fig 2.9 when again left arrow key was pressed to select lower view port and ‘s’ key was pressed a triangle was drawn in lower viewport leaving upper view port unaffected.



**Fig 2.10 Displays the output when wired tea pot in selected viewport is replaced via square**

In fig 2.10 when right arrow key was pressed upper view port get selected and then by pressing ‘a’ key wired tea pot was drawn earlier was replaced with a square in upper view port leaving lower view port unaffected.

**Bibliography**

1. <https://www.cs.cmu.edu/~fp/courses/graphics/pdf-color/10-texture.pdf-> M. Ian Graham
2. <http://www.cs.cmu.edu/~./ph/texsurv.pdf-> Paul S. Heckbert
3. <http://web.eecs.utk.edu/~huangj/cs456/notes/456_texturemap1.pdf-> Jian Huang