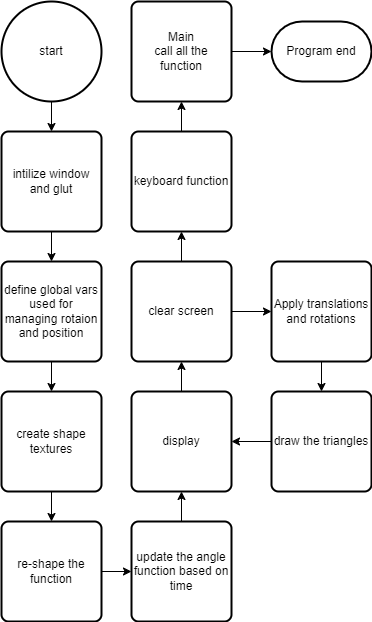
**Putting It All Together**

By Benjamin Carter and Josh Canode

**Part 1**

**Programming Implementation and Processing Codes and Flowchart**

* Start
* Initialize GLUT
  + Initialize window size and properties
  + Register callback functions
  + Create a window
* Set up global variables (x\_pos, y\_pos, zoom, angle, rotating)
* Define a red and yellow checkered texture
  + Set up the reshape function
  + Set the viewport
  + Set the projection matrix
* Set up the texture
* Set up the updateAngle function
  + Update the rotation angle
  + Call the function recursively with a timer
* Set up the display function
  + Clear the screen
  + Set up the camera position and orientation
  + Apply translations and rotations
  + Draw the textured triangles
* Set up the keyboard function
  + Handle keypress events
  + Stop or continue the rotation
  + Zoom in or out
* Set up the specialKeys function
  + Handle special key events
  + Move the camera position (up, down, left, right)
* Enter the main loop
* Call the display function to render the scene



**Mesh**

We did not use a mesh in part 2. This is because the objects drawn were cubes/planes. The planes were just 3D polygons with four vertex points.

**Keyboard Interface**

A screen shot of a computer

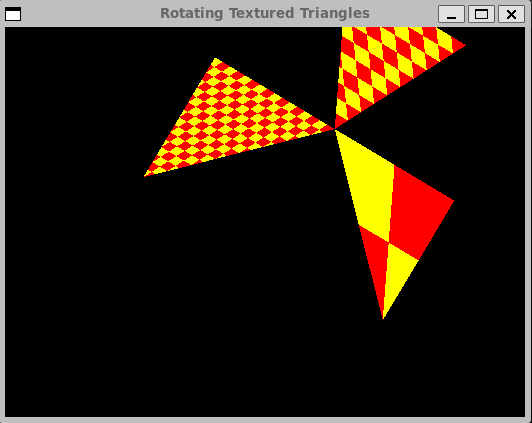
Description automatically generated

1. Use 's' to stop animation
2. Use 'c' to continue animation
3. Use '-' to zoom out
4. Use '+' to zoom in
5. Use arrow keys to move object up/down/left/right

**Screenshots**

A screenshot of a computer

Description automatically generated



**Part 2**

**Mathematical Concepts**

In this program, there are several mathematical principles involved. First is using trigonometry to adjust the camera position. The camera is adjusted with the following parametric equation:

This creates an elliptical orbit around the center on the yz plane.

Also, when objects bounce off of a plane, the velocity is “reflected” around the normal vector of the plane.



In this drawing, an object comes in contact with a plane, and then its velocity vector is reflected around the normal of the plane. This is how the cube “bounces”. It uses the glm::reflect() function to calculate the new vector.

Third, linear algebra is used to convert vertices from one coordinate space to another. Multiplying a vertex by a change-of-base matrix allows that vertex to be transformed to a new location/rotation while still maintaining relative direction and magnitude. This allows for one change such as reflection of velocity to be applied to many vectors.

A computer code on a black background

Description automatically generated

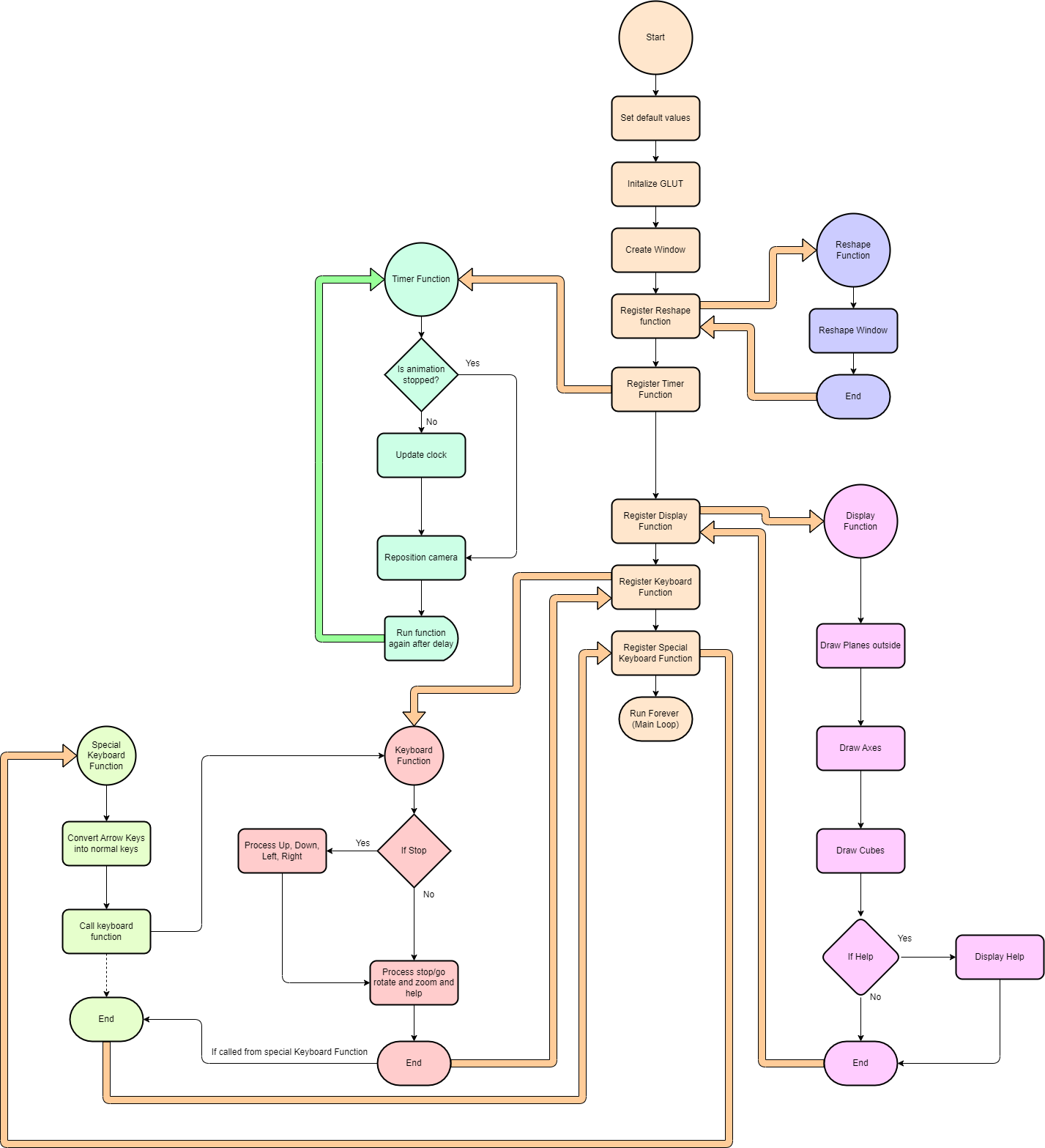
This shows the use of linear algebra to get the transformed vector *result*.

**Programming Implementation and Processing Codes and Flowchart**

The program uses multiple functions:

1. mapValue
   1. Mathematical function for scaling values
2. CubeInfo stuct
   1. Constructor
   2. brightnessAdd – do color balance on cubes
3. Cube::drawCube
   1. Draw a cube on the screen
4. renderBitmapString
   1. Render text onto a screen
5. StringDraw
   1. Used to render text to the screen
6. draw
   1. The main drawing loop. Draws axis, planes, and cubes.
7. display
   1. This calls draw and then flushes display
8. Timer
   1. This function runs at a rate of 60Hz. This is the “frame” script.
9. Reshape
   1. This function is called when the window is reshaped. It adjusts the perspective/camera view settings
10. Init
    1. This function initializes aspects of OpenGL
11. Keyboard
    1. This function handles keyboard input.
12. placeholderKeyboard
    1. This function handles “special keys”. It translates special keys to “simple” keys to then pass along to the keyboard function.
13. Main
    1. This function runs when the program is first called.

See flowchart for implementation and interaction of functions.



**Mesh**

We did not use a mesh in part 2. This is because the objects drawn were cubes/planes. The planes were just 3D polygons with four vertex points.

**Keyboard Interface**

A screenshot of a computer

Description automatically generated

1. Use ‘s’ to stop the animation.
2. Use ‘c’ to continue the animation.
3. Use ‘r’ to rotate the animation. Does not require it to be stopped first
4. Use ‘u’ or the up-arrow to move the image up. Requires the animation to be stopped first
5. Use ‘d’ or the down-arrow to move the image down. Requires the animation to be stopped first.
6. Use ‘o’ or the left-arrow to move the image left. Requires the animation to be stopped first.
7. Use ‘p’ or the right-arrow to move the image right. Requires the animation to be stopped first.
8. Use ‘+’ to zoom in. Use ‘-‘ to zoom out. Or, use ‘[‘ and ‘]’ respectively.

**Animation**

There are three animations running at the same time in the program.

1. Each individual cube is rotating over time
2. The cubes are bouncing between both planes
3. The “camera” of the scene orbits around the center and is customizable.

The cube rotation animation is done through rotation transformations using the glm library

|  |
| --- |
| glm::mat4 model = glm::mat4(1.0f);      model = glm::translate(model, pos);    model = glm::rotate(model, timeU, glm::vec3(sin(timeU), cos(timeU),  sin(timeU / 3))); |

The cubes are bouncing between both planes using glm::reflect()

|  |
| --- |
| glm::vec3 pos = cbInfo->velocity \* (timeU - cbInfo->oldTime) +  cbInfo->oldPosition;    if((pos.x > planeX && cbInfo->velocity.x > 0) ||  (pos.x < -planeX && cbInfo->velocity.x < 0))    {        glm::vec3 normalPlane = glm::vec3((pos.x > 0) \* -2 + 1, 0, 0);        glm::vec3 ref = glm::reflect(cbInfo->velocity, normalPlane);        cbInfo->velocity = ref;        cbInfo->oldPosition = pos;        cbInfo->oldTime = timeU;    } |

The camera of the scene orbits the center:

|  |
| --- |
| void timer(int v) {    if(!stop)    {      timeU += 0.01;    }      glLoadIdentity();      glm::mat4 model = glm::mat4(zoom);      glm::vec3 normal = glm::vec3(0, -10 \* cos(timeU), -30 \* sin(timeU));      model = glm::rotate(model, rotate, normal);      glm::vec4 result = model \* glm::vec4(0.0f, -10 \* sin(timeU), 30 \*  cos(timeU),0.0f);        result = result + camOffset;      gluLookAt(result.x,result.y,result.z, camOffset.x, camOffset.y,  camOffset.z, normal.x, normal.y, normal.z);      glutPostRedisplay();      glutTimerFunc(1000/60.0, timer, v);  } |

**Screenshots**

A computer screen shot of a computer screen

Description automatically generated

A computer screen shot of a computer screen

Description automatically generated

A screenshot of a computer

Description automatically generated

**References**

We did not use any external sources other than the given class starting code.