Implementation of LTSGraph3D

Ali Deniz Aladagli

April 26, 2024

Contents

1 Introduction

The ltsgraph tool allows the visualization of a given labeled transition system, an LTS, as a graph and the manipulation of its layout. These graphs are state spaces, where every transition is directed and has a label. The ltsgraph tool visualizes this state space only in two dimensions. The LTSGraph3D tool discussed in this report visualizes a graph of a LTS in three dimensions. The changes in classes with respect to the implementation of ltsgraph are discussed through sections 2 to 8.

2 Utils namepace

The Utils namespace provides the LTSGraph3D tool with some general utilities that are required in the program, such as a combined 3D position variable, vector and matrix operations. The rotation matrix generating function is also defined here.

2.1 Utils::Vect

The Utils::Vect is a structure that defines three double precision numbers that are either used to represent the x, y and z coordinates of an object or a vector in 3D space. It is used by the State class to represent the positions of the states. It is also used in the matrix normalization in the GLCanvas class.

2.2 vecLength

The vecLength method returns the length of the vector by using the Pythagoras theorem in 3D.

2.3 angDiff

Given two vectors the angDiff function returns the angle between these two vectors in radians, by using the fact that the dot product of two vectors is equal to the product of the two vectors' lengths multiplied by the cosine of the angle between them. Given the length found by Utils::vecLength and the dot product found by Utils::dotProd, this angle can be found by using the arccos function. To prevent errors, produced by the lack of precision of double variables, the value of the input to the arccos function is always provided in the [-1,1] interval where the function is defined.

2.4 dotProd

Given two vectors, the dotProd function returns their dot product.

2.5 MultGLMatrices

The MultGLMatrices function takes in two arrays of size 16 that represent two 4×4 matrices that are lined up according to their notation in OpenGL. A matrix

$M_{4\times4}$ would be represented by an array A of size 16 as below:

$$\begin{pmatrix} A_0 & A_4 & A_8 & A_{12} \\ A_1 & A_5 & A_9 & A_{13} \\ A_2 & A_6 & A_{10} & A_{14} \\ A_3 & A_7 & A_{11} & A_{15} \end{pmatrix}$$

Using this notation, this method multiplies the first matrix by the second and returns the answer as the third parameter.

2.6 genRotArbAxs

Given an angle and the normal of the axis to be rotated about, the genRotArbAxs function generates a rotation matrix.
To create a rotation matrix the unit vector of the rotation axis is needed. According to right handed rotations used in OpenGL the unit vector of the rotation matrix can be found as below:

$$\alpha = arctan\left(\frac{y}{x}\right)\ ,\ U_x = -sin(\alpha)\ ,\ U_y = cos(\alpha)$$

Where x and y are the x and y components of the input normal vector and U_x and U_y are the x and y components of the unit vector $\vec{U}$ of the rotation axis. Since rotation around the z-axis is not needed in the program, calculations involving the z component is ignored. A rotation matrix around an arbitrary axis given by its unit vector can be generated as below:

$$\begin{pmatrix} (1-c)*U_x^2+c & (1-c)*U_x*U_y+s*U_z & (1-c)*U_x*U_z-s*U_y & 0 \\ (1-c)*U_x*U_y-s*U_z & (1-c)*U_y^2+c & (1-c)*U_y*U_z+s*U_x & 0 \\ (1-c)*U_x*U_y+s*U_y & (1-c)*U_y*U_z-s*U_x & (1-c)*U_z^2+c & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Where c is cosine of the angle to be rotated, s is the sinus of the angle to be rotated and U_z is the z coordinate of the unit vector of the axis. However in the implementation used in this tool, U_z is always taken as zero.

2.7 GLUnTransform

Given a modeling and viewing transformation matrix, the GLUnTransform function reverses these transformations for a given vector. To do this, the following facts are used:

- The inverse of a rotation matrix is equal to its transpose, since the determinant of a proper rotation matrix is equal to 1.
- To reverse effects of a modeling and viewing transformation matrix, the inverse of that matrix is required.

Since a rotation matrix is itself a modeling and viewing transformation matrix, a prerequisite that the input matrix is a proper rotation matrix lets us use these two facts. Even if the input matrix contains translations in the rightmost column, it can be ignored since when transposed, this column will only affect the fourth coordinate of the input vector (which defines how it will be affected by

further translations, but will not be used in this program again). The transpose of the input 4×4 and 4×1 coordinates to be reversed are multiplied using matrix multiplication. And the first three elements of the answer are our x, y and z coordinates respectively.

3 State

The State class is used to represent the states. The only change from ltsgraph is that this class now uses the Utils::Vect structure, now defined in 3D, in order to represent the state's location in the three dimensional coordinate system. The constructor is changed accordingly.

4 Transition

The Transition class is used to represent transitions. Each transition has a label and a handle (drawn as a little square). The line drawn to represent the transition always goes through this handle and the user can click on this handle and move it to curve the transition. The main changes in the Transition class, apart from adding the z coordinate to the label positions, focuses on the coordinates of the handles of the transitions. In both ltsgraph and LTSGraph3D, when a handle is moved and given a new position, this position is kept by how it differentiates from its original position. The original position of a handle is mid-point of the origin and the target states for non-self loop transitions, which creates a straight line. While in 2D keeping one angle is enough to clarify this difference, in 3D three angles are needed.
If there is a transition between an origin state O, a target state T and the new coordinates of the handle are defined by the vector $\vec{H}$, the parameters needed to recalculate $\vec{H}$ are as below:

$$\vec{TRA} = \vec{T} - \vec{O}$$
$$\vec{HAN} = \vec{H} - \vec{O}$$
$$control\alpha = arctan\left(\frac{\vec{HAN}_x}{\sqrt{(\vec{HAN}_y)^2+(\vec{HAN}_z)^2}}\right) - arctan\left(\frac{\vec{TRA}_x}{\sqrt{(\vec{TRA}_y)^2+(\vec{TRA}_z)^2}}\right)$$
$$control\beta = arctan\left(\frac{\vec{HAN}_y}{\sqrt{(\vec{HAN}_x)^2+(\vec{HAN}_z)^2}}\right) - arctan\left(\frac{\vec{TRA}_y}{\sqrt{(\vec{TRA}_x)^2+(\vec{TRA}_z)^2}}\right)$$
$$control\gamma = arctan\left(\frac{\vec{HAN}_z}{\sqrt{(\vec{HAN}_y)^2+(\vec{HAN}_x)^2}}\right) - arctan\left(\frac{\vec{TRA}_z}{\sqrt{(\vec{TRA}_y)^2+(\vec{TRA}_x)^2}}\right)$$
$$controlDistance = \frac{|\vec{HAN}|}{|\vec{TRA}|}$$

Where $\vec{TRA}$ is the vector that represents the straight line between the origin and the target states. $\vec{HAN}$ is the vector that represents the straight line between the origin state and the new handle position. $control\alpha$, $control\beta$ and $control\gamma$ are the angles that we need to keep and $controlDistance$ is the proportion needed to calculate where the handle would stand in the direction of $\vec{HAN}$. These angles are the decompositions of the angle between $\vec{TRA}$ and $\vec{HAN}$ into three angles with three planes (y-z, x-z and x-y respectively).
For a self-loop transition the difference of the handle point with respect to the state is kept in terms of these angles:

$$control\alpha = arctan\left(\frac{H\vec{A}N_x}{\sqrt{(H\vec{A}N_y)^2+(H\vec{A}N_z)^2}}\right)$$
$$control\beta = arctan\left(\frac{H\vec{A}N_y}{\sqrt{(H\vec{A}N_x)^2+(H\vec{A}N_z)^2}}\right)$$
$$control\gamma = arctan\left(\frac{H\vec{A}N_z}{\sqrt{(H\vec{A}N_y)^2+(H\vec{A}N_x)^2}}\right)$$
$$controlDistance = \frac{|H\vec{A}N|}{200}$$

Where $H\vec{A}N$ is defined as it is for non-self looping transitions.
The reason the handle position is kept in this way is to guarantee that the handle moves consistently when the origin or the target states move. In this way the angle of the curve is preserved.

5 LTSGraph3D

The LTSGraph3D class contains the entry point to the program. Nearly all of the implementation is the same as ltsgraph. Having access to nearly all the other classes, the following methods are implemented here.

5.1 LTSGraph3D::moveObject

In LTSGraph3D some of the drawn objects can be moved selecting them with the left mouse button. These objects can either be a state, a transition handle or a transition label. The LTSGraph3D::moveObject method moves the selected object according to the current modeling and viewing transformations, so that the object always moves the same direction of the mouse movement and by the same magnitude. This enables moving an object in three dimensions. To do this we assume that the movement on the screen is the transformed version of the movement that should have been done internally. Reversing the transformation on the movement input with Utils::GLUnTransform will give us the internal movement that is kept. This allows us to apply the proper direction to the movement. We still have to convert the screen coordinates to the way they are stored in the Graph class and perform reverse the projection so that objects in any depth stay under the mouse while moving them. We can do the screen to storage conversion by reversing the calculations done while they are drawn in Visualizer::visualize:

$$rad = radius * pixelSize$$

rad is equal to the *radius* used for the states returned by Visualizer::getRadius multiplied by the *pixelSize* (returned by the method GLCanvas::getPixelSize). Pixels are the units used for screen coordinates.

$$pixelToWorld = 3 * 550/pheight$$

pixelToWorld is a coefficient for screen to world coordinates conversion found by trial and error for an object moving in the world y-axis at a constant certain depth. 550 is the window height in pixels the trials were done, *pheight* is the height in pixels of the current window, used to make movement consistent with any window size and 3 is the constant found through these trials.

$$x' = x * 2000/(width - rad * 2) * pixelToWorld$$
$$y' = y * 2000/(height - rad * 2) * pixelToWorld$$
$$z' = z * 2000/(depth - rad * 2) * pixelToWorld$$

x, y and z are the untransformed pixel coordinates, *width*, *height* and *depth* are the world coordinates of the dimensions of the window, as they are returned by the method GLCanvas::getSize and 2000 is the constant used for the conversion between world coordinates and storing coordinates in Visualizer::visualize. x', y' and z' are the new world coordinates yet to be modified to the correct magnitude.
To apply the correct magnitude, we need the depth of the object we want to move in world coordinates (where it is drawn). Depth is defined with respect to the local coordinate system of the graph and is different then the z coordinate kept in the object if any rotations are done. To do this we translate to the coordinates of the object, just like we were drawing it. At this point we know that the 14^{th} element of the modeling and viewing matrix (with OpenGL notation), that occurs after the translation, is the real depth where the object was drawn in world coordinates. For an object, this value always has to be smaller then zero because only objects 'inside' the screen are drawn (in OpenGL, inside the screen is negative z axis and the user's side of the screen is positive z axis). Since projection is directly proportional with the depth, we can directly multiply the above real depth with the storing coordinates calculated above (x', y' and z').

$$x'' = x' * depth\ ,\ y'' = y' * depth\ ,\ z'' = z' * depth$$

By using the object's setters we can directly add these values (x'', y'' and z'') found to their current coordinates.

5.2 LTSGraph3D::getCanvasMdlvwMtrx

This method returns the current modeling and viewing transformation matrix, for the classes that can access the LTSGraph3D class but cannot access the GLCanvas class, using GLCanvas::getMdlvwMtrx.

5.3 LTSGraph3D::getCanvasCamPos

This method returns the current camera position, for the classes that can access the LTSGraph3D class but cannot access the GLCanvas class, using GLCanvas::getCamPos.

6 GLCanvas

As in the tool ltsgraph, canvas operations such as projection and object picking are all implemented in the GLCanvas class, which is an extension of the wxGLCanvas class defined in wxWidgets. In initialization depth testing is enabled, meaning that the objects that are behind others will not be drawn.

6.1 GLCanvas::display

The perspective projections that are needed to apply the 3D effect are done by calls to gluPerspective, which is a part of the glu library. The far clipping plane

distance is selected so that after any rotations, the whole graph is still visible. To be able to visualize the graph in any way the user wants, features such as rotations, panning and zooming are implemented. To implement these features modeling and viewing transformations are needed. For ease of other calculations these transformations are implemented such that the camera is always in the center of the world coordinates and the model is rotated or moved instead of the camera. Rotations are implemented as the rotations of an arc ball to avoid problems such as the gimbal lock (which may cause unexpected rotations). An arc ball rotation is one which always occurs in planes produced by the world axises (which never change) instead of the planes produced by the local axises of the model (which change with every new rotation). Since the rotations that the OpenGL library uses are all done in the local planes, functions to create and multiply rotation matrices are implemented the Utils namespace. To achieve an arc ball rotation the method used in this tool is to apply every new rotation before applying the previous modeling and viewing transformations, which are represented with a 4×4 matrix, however implemented as an array of size 16 to provide compatibility with the OpenGL library. While a rotation tool is being used, every mouse movement defines a vector. The magnitude of this vector is the angle to be rotated and the direction is the normal to the axis that is being rotated around. The z component of this vector is always zero since we do not have a usable way of mapping mouse movement to third dimension. Giving these values as inputs to the method Utils::genRotArbAxs, the needed rotation matrix is generated. After this 4×4 matrix is generated it is only a matter of multiplying this matrix with the previous modeling and viewing matrix by using Utils::MultGLMatrices:

$$MVT'_{4\times4} = RM_{4\times4} \times MVT_{4\times4}$$

Where $RM_{4\times4}$ is the rotation matrix generated by the formula above and $MVT_{4\times4}$ is the modeling and viewing transformation matrix kept. Even though variables with double precision are used in these formulas, some accuracy is always lost during these operations. These losses result in sheering and stretching in the graph drawn. Therefore after a rotation, $MVT_{4\times4}$ is normalized by the method GLCanvas::normalize matrix, so that these errors can be minimized. In this function OpenGL's rotation functions are used on a matrix which is initially an identity matrix to recreate a rotation matrix without sheering and stretching that is closes to the matrix to be normalized. This matrix can be represented by $N_{4\times4}$ This normalizing function works by the idea that $MVT_{4\times4}$ stores the unit vectors for the three axises (local axises of the graph) generated after rotations in world coordinates. With OpenGL's notation of matrices (any translations are ignored):

$$\vec{X} = \{MVT_{0,0}, MVT_{1,0}, MVT_{2,0}\}\ ,\ \vec{Y} = \{MVT_{0,1}, MVT_{1,1}, MVT_{2,1}\}\ ,$$
$$\vec{Z} = \{MVT_{0,2}, MVT_{1,2}, MVT_{2,2}\}$$

Where $\vec{X}$, $\vec{Y}$ and $\vec{Z}$ are these unit vectors in world coordinates. Firstly the unit vector for the z-axis ($\vec{Z}$) is assumed to be in the correct direction. Two rotations are needed to overlap $N_{4\times4}$'s z-axis with $\vec{Z}$. The first rotation is around the local y-axis of $N_{4\times4}$ and this rotation's angle is the angle between world z-axis and $\vec{Z}$'s projection on the x-z plane. The second rotation is around the local x-axis of $N_{4\times4}$ and this rotation angle is the angle between $\vec{Z}$ and its projection on the x-z plane. Calculations are as below:

$$\beta = arctan\left(\frac{MVT_{0,2}}{MVT_{2,2}}\right) \ , \ \gamma = arctan\left(\frac{MVT_{1,2}}{\sqrt{MVT_{0,2}^2+MVT_{2,2}^2}}\right)$$

Where β and γ are respectively the first and the second angles. With two calls to glRotate in the OpenGL library, the new z-axis is pointing to the correct direction with minimal stretching or sheering. At this step a second assumption is made that the projection of the deformed y-axis, represented by its unit vector $\vec{Y}$, on the local x-y plane is the correct direction for the new y-axis. Projecting allows us to compute a correct angle, between $\vec{Y}$ and the y-axis of $N_{4\times4}$, to rotate around the local z-axis of $N_{4\times4}$. The normal we use to project is nothing but the z-axis of $N_{4\times4}$.

$$\vec{Y_{projection}} = \vec{Y} - ((\vec{Y}.\vec{normal}) \times \vec{normal})$$

Where . is the dot product between two vectors and is implemented in Utils namespace as Utils::dotProd. Utils::angDiff also from the Utils namespace is used to calculate the angle between $\vec{Y_{projection}}$ and the local y-axis of $N_{4\times4}$. However, there is an ambiguity in the angle found (since cosine function used to find the angle difference is a modulo π function). To resolve this ambiguity the y coordinate of the $\vec{X}$ is looked at. If this value is negative the glRotate function is called with a negative rotation around the local z-axis of $N_{4\times4}$ and with a positive rotation otherwise. After these rotations the matrix kept within the current OpenGL context is the fixed rotation matrix that represents all the rotations done previously. Now we can replace our $MVT_{4\times4}$ with this matrix. After doing these calculations, panning and zooming effects are fairly easy:

$$MVT_{0,3} = -lookX \ , \ MVT_{1,3} = -lookY \ , \ MVT_{2,3} = -lookZ$$

Where $lookX$ is the panning done in the x-axis, $lookY$ is the panning done in the y-axis and $lookZ$ is the total zooming done. For a more realistic display, a light that always comes from the direction of the camera is added however not enabled yet. This is done before loading the modeling and viewing transformation matrix to OpenGL so that the light's direction is not affected by the transformations. To be able to color the states, a material behavior is also added. After the matrix is loaded the control is given to Visualizer::visualize to draw the LTS to the screen. When this function returns, upon the user's request, a representation of the local coordinate system of the model can be shown at the left lower corner of the window in a new viewport. To make it more visible, this coordinate system is not affected by the panning and zooming effects.

6.2 Mouse Functions

All mouse functions are initiated by the event handler of wxWidgets. Mouse movement is processed by the method GLCanvas::onMouseMove, only has a function if one of mouse buttons is clicked. These functions are as below:

- Left Mouse Button
  Defined by the GLCanvas::onMouseLftDown, the left mouse button can have up to two functions, one active when an object is clicked and the other active otherwise. When not clicking on an object, the left mouse can either be used for panning, zooming and rotating or this tool can be completely disabled to prevent unwanted changes in the viewpoint.

When clicked on an object either the state's color is changed with the active color or the state can be initiated to move as long as the left mouse button is held, this tool cannot be disabled. These tools are defined by their identification numbers, defined in the IDS namespace, extended after the last identification number used by wxWidgets. To check if an object is clicked or not the type of the method GLCanvas::pickObjects is changed to boolean from void. To move an object, new mouse coordinates are passed to LTSGraph3D::moveObject since the depth of the selected object has to be known to make mouse movement equal to object movement and GLCanvas class does not have access to the selected objects.

- Right Mouse Button
  The right mouse button has two functions. If a state is clicked, that state's position is stabilized and it can not be moved by the layout algorithm implemented, only the user can move it. If empty space is clicked this button always does a rotation.

- Middle Mouse Button
  To increase usability for mouses with a wheel/middle button, zooming can be done by turning the wheel (GLCanvas::onMouseWhl) and rotation can be done by clicking the middle button anywhere in the window and moving the mouse (GLCanvas::onMouseMidDown).

When one of these functions are being used the mouse cursor is changed by the method GLCanvas::setMouseCursor to a proper cursor for the function, again defined by the active tools identification number.

6.3 GLCanvas::pickObjects

To pick objects on the screen, the rendering mode is switched to selecting mode. A picking region is created around the click location. In this mode the 'names' that are given to an object when they were being drawn are recorded to a buffer when they are in the picking region defined by the mouse click (called a hit). With these names the object selected can be defined. If hits are not bigger then zero, no object is selected. Only one object can be selected at a time. Clicking on empty space or a new object clears the selection on the previous object. The selected objects are pointed to by LTSGraph3D class and their boolean selected check value is changed to true by the Graph class.

6.4 GLCanvas::getCamPos

The GLCanvas::getCamPos method is implemented to be able to get the camera position if it were thought as the camera being moved instead of the graph.

6.5 GLCanvas::getSize

The GLCanvas::getSize method defines the height and the width in world coordinates just as in the same named method in ltsgraph. The depth is defined as their arithmetic mean to provide balanced sizes in the environment.

6.6 GLCanvas::getMdlvwMtrx

The GLCanvas::getMdlvwMtrx method is implemented to supply the other classes with the transformations done for numerous purposes.

7 Visualizer

All of the drawings are done in this class as it is in the ltsgraph tool.

7.1 Visualizer::drawStates

The Visualizer::drawStates method initiates the drawing of the objects. For every state, its out going transitions and self loops are drawn. The rendering process of the texts are separated since the texts cannot be rendered without disabling the depth mask (which disables the writing done to the depth buffer used for depth testing), therefore they should be rendered after everything else is rendered. Therefore they are taken care of in separate loops. After every state and transition are drawn by calls to the methods Visualizer::drawState and Visualizer::drawTransition (Visualizer::drawSelfLoop for self loops) respectively, the texts are rendered using Visualizer::drawTransLabel and Visualizer::drawStateText methods. All of these four methods convert the storing coordinates to world coordinates using the calculations below:

$$rad = radius * pixelSize \; x' = (x/2000) * (width - rad * 2)$$
$$y' = (y/2000) * (height - rad * 2) \; z' = (z/2000) * (depth - rad * 2)$$

rad is equal to the *radius* used for the states kept in the Visualizer class multiplied by the *pixelSize* returned by the method GLCanvas::getPixelSize. *width*, *height* and *depth* are the world coordinates of the dimensions of the window, as they are returned by the method GLCanvas::getSize and 2000 is the constant used for the conversion between world coordinates and storing coordinates. These calculations are necessary for consistent rendering in different window sizes and different aspect ratios.

7.2 Visualizer::drawState

To draw the states and the selection notifying borders around them, gluSphere and gluPartialDisk from the glu library are respectively used. To make the borders visible from any angle, the modeling and viewing transformation matrix is only applied to the translation part of the ring to find its location in the world coordinates with the disk always facing the user. After its border is drawn, to draw the state lighting has to be enabled. The spheres being the only 3D object that needs lighting, it is only enabled at this part of the code. After two calls to glPushName make the state selectable by the user, the sphere is drawn after being translated to its location with the radius *rad* calculated above. Lighting is again disabled to that drawings such as transitions are not affected by it.

7.3 Visualizer::drawTransition

Second order Bezier curves using one control point are used to draw the transitions, as they are in ltsgraph. The transition handle (the virtual control point)

lies in the mid-point of this one control point and the mid-point of the 2 states that the transition is tied to. After the transition is drawn using Bezier equations, if displaying the transition handles is enabled, a white cube with colored sides (depending on selection) is drawn after using glPushName to make this handle selectable. To indicate the direction of the transition, small cones are drawn in the correct location and pointing to the correct direction using Visualizer::drawArrowHead. First we translate to the point where the target sphere was drawn. Since we know that Visualizer::drawArrowHead draws the cones pointing outwards from the current local z-axis, to make the cone point to the direction we want, we have to overlap this direction with the positive z-axis. We want the direction of the cone lying from the real control point described above to the center of the sphere it is heading to. We can realize this with two rotations using glRotate:

$$\beta = arctan\left(\frac{x_{To}-x_{Control}}{z_{To}-z_{Control}}\right),\ \gamma = arctan\left(\frac{y_{To}-y_{Control}}{\sqrt{(x_{To}-x_{Control})^2+(z_{To}-z_{Control})^2}}\right)$$

The first rotation is around the local y-axis using the angle β. The second rotation is around the local x-axis using the angle $-\gamma$. After these two rotations, by translating $2 \times rad$ in the local minus z-axis, we draw the cone pointing to the sphere instead of inside it.

7.4 Visualizer::drawSelfLoop

Third order Bezier curves with 2 control points are used to curve the transition back to its looping state, as it is in the ltsgraph tool. Since these transitions have only one handle each, the self loops existing plane cannot be rotated around the axis created by the state and the one handle, without moving the one handle. Therefore, for simplicity, the two control points always stand in the same depth (in the local coordinate system of the graph). Instead of using the angle between the axis that goes through the transition's from state and the handle and the x axis as in ltsgraph, here the angle between the axis that goes through the transition's from state and the handle and the x-y plane is used to calculate the x and y coordinates of the two control points. Notice that the sinus of both angles gives the y coordinate of the handle. To keep the distances of the control points from the looping state the same while adding the z coordinate, we calculate the z coordinate in a similar way the x and y coordinates of the control points are calculated:

$$zFactor = (4 * (zVirtual - zState))/(3 * (sin(\gamma)))zControl = zState + zFactor * sin(\gamma)$$

Where γ is the angle between the axis that goes through the transition's from state and the handle and the x-y plane, returned by Transition::getControlGamma. $zState$ is the z coordinate of the looping state, $zVirtual$ is the z coordinate of the handle. $zControl$ is the depth of both control points. This calculation is only done when $|sin(\gamma)|$ is greater then 0.01, since otherwise the calculation above would be undefined. If it is smaller then 0.01, this means the difference of angle is not really great and we can directly take the value of the z coordinate as same as the z coordinate of the looping state.

7.5 Visualizer::drawArrowHead

Assuming the proper translations and rotations are done, the Visualizer::drawArrowHead function draws a cone (a cylinder with top radius equal to zero) with the input height and base radius equal to the one fifth of this height. The cones are drawn with the gluCylinder function in the glu library. The drawn cone points towards the local positive z-axis.

8 Springlayout

Like in ltsgraph, when enabled the Springlayout class tries to find an optimized, stable layout using a force-directed placement where the goal is to find an equilibrium state for every state, where the sum of the forces applied on it is 0. Vertexes function as electrically same charged particles which repel each other more as they get closer. Edges function as springs that exerts force on the vertexes connected to them depending on their current length and their zero-energy length. If the length of an edge is smaller then its zero-energy length, it exerts repulsive force and exerts attractive force otherwise.

8.1 Springlayout::layoutGraph

The algorithm works exactly the same way in three dimensions as it works in two dimensions. However the walls that keep the graph limited to the window that are implemented in ltsgraph are removed here, since we can move, zoom or rotate to anywhere in the viewing space. We have to add the third coordinate and integrate the required force formulas into 3D. The formulas for the electric and spring forces are given below:

$$\vec{F}^{electric}(u,v) = \frac{e}{\vec{d}(u,v)^2}$$

$$\vec{F}^{spring}(u,v) = s * \log\left(\frac{\vec{d}(v,u)}{l}\right)$$

Where $\vec{F}^{electric}(u,v)$ is the repulsion force exerted to a vertex v by another vertex u. $\vec{F}^{spring}(u,v)$ is the force exerted to a vertex v by a spring that is connected to the vertex u. $\vec{d}(u,v)$ is the distance between vertexes u and v,e is the equal charge coefficient of all vertexes, s is the equal stiffness coefficient and l is the equal zero-energy length of all edges. We add the z component of the gravitational force according to the imaginary screen depth defined in the implementation:

$$F_x^{gravitation}(v) = (-2) * x_v/2000$$

$$F_y^{gravitation}(v) = (-2) * y_v/2000$$

$$F_z^{gravitation}(v) = (-2) * z_v/2000$$

We can find all three components of the total forces applied on a vertex v as below:

$$F_x^{total}(v) = \sum_{u \in V} \left(\|\vec{F}^{electric}(u,v)\| * \frac{x_v - x_u}{\|\vec{d}(u,v)\|} \right) + \sum_{(u,v) \in E} \left(\|\vec{F}^{spring}(u,v)\| * \frac{x_v - x_u}{\|\vec{d}(u,v)\|} \right) + F_x^{gravitation}(v)$$

$$F_y^{total}(v) = \sum_{u \in V} \left(\|\vec{F}^{electric}(u,v)\| * \frac{y_v - y_u}{\|\vec{d}(u,v)\|} \right) + \sum_{(u,v) \in E} \left(\|\vec{F}^{spring}(u,v)\| * \frac{y_v - y_u}{\|\vec{d}(u,v)\|} \right) + F_y^{gravitation}(v)$$

$$F_z^{total}(v) = \sum_{u \in V} \left(\|\vec{F}^{electric}(u,v)\| * \frac{z_v - z_u}{\|\vec{d}(u,v)\|} \right) + \sum_{(u,v) \in E} \left(\|\vec{F}^{spring}(u,v)\| * \frac{z_v - z_u}{\|\vec{d}(u,v)\|} \right) + F_z^{gravitation}(v)$$

The mass of the vertex can be thought as one and the acceleration of that vertex will be equal to the total sum of the forces on the vertex; where the velocity of the vertex is not conserved and will be zero at the beginning of each iteration. Therefore, the new coordinate of a vertex is found directly by adding the x-component of the current sum of the forces to the current x coordinate of the vertex, adding the y-component of the current force to the current y coordinate of the vertex and adding the z-component of the current force to the current z coordinate of the vertex. The new coordinates of the vertex v are as below:

$$(x_v^{'}, y_v^{'}, z_v^{'}) = (x_v + F_x^{total}(v), y_v + F_y^{total}(v), z_v + F_z^{total}(v))$$