HBM 601E Computational Geometry

Homework 3 Report

Kadriye Tuba Turkcan

29.01.2020

Introduction:

This report represents an interactive graphics program that performs k th order polynomial approximation and creates porcupines for each approximated polynomial. It also interpolates the given points using both parametric Lagrange Polynomial Method and Improved Parabolic Blending Method using OpenGl in C programming language.

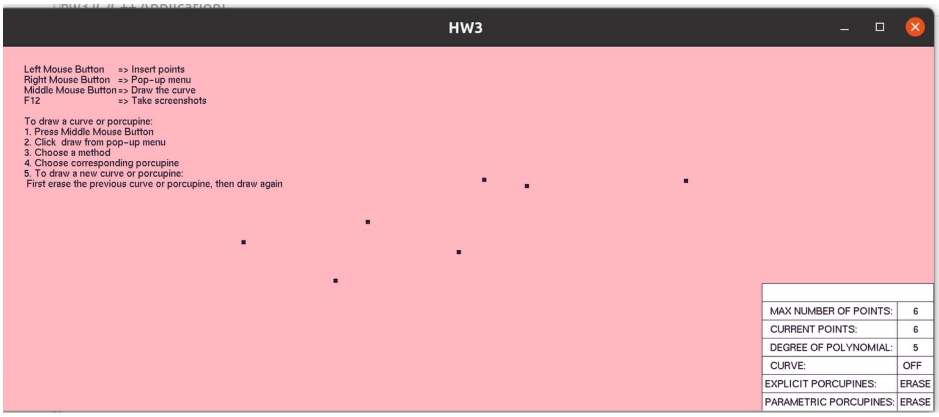
First, number of points (N_max) are asked from the user by inserting from the console. Number of points gives us the order of the polynomial (M). 1 less than this value is the degree of our polynomial. Then, user will be able to draw the points on the given screen using mouse.

window.h:

Inside “window.h” header file, we define the size of the window with `windowsizeX` and `windowsizeY`. The size is set to 1800 and 700 in the menus.c file, for X and Y components, respectively.

menus.h :

The user sees the following screen first after clicking the points. The right bottom menu , left top instructions and a pop-up menu are given inside menu source file. On the left top corner of the window, there are instructions for the user to follow.

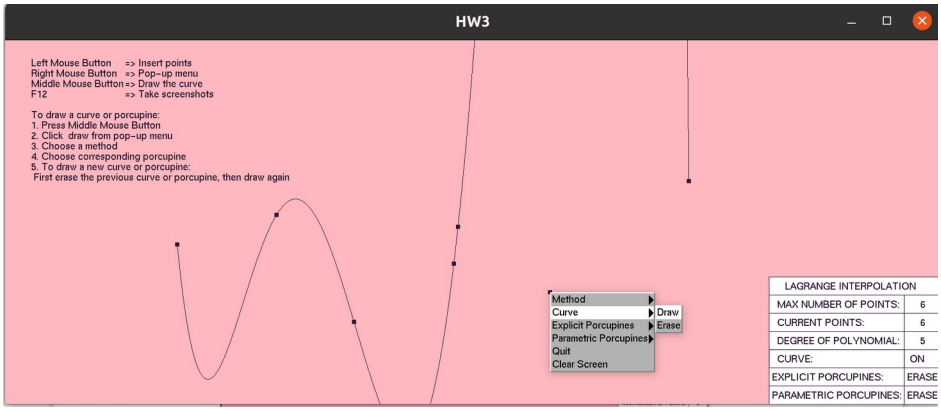


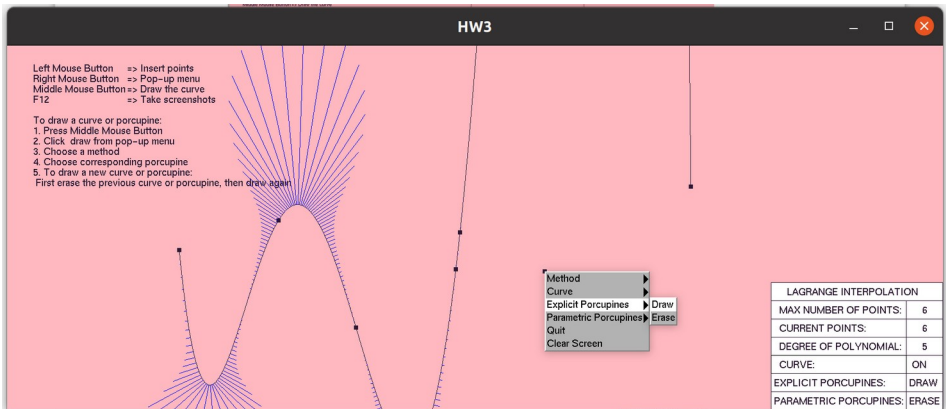
These instructions are given with the instructions() function:

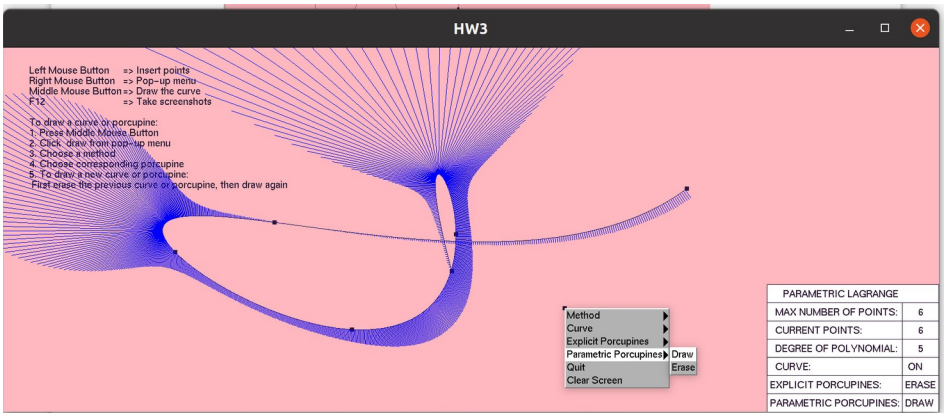
```
void instructions(void)
{
	glColor3f(0.19, 0.1, 0.21 );
	drawBitmapText("Left Mouse Button",50.0,50.0 );
	drawBitmapText("=> Insert points",230.0,50.0 );
	drawBitmapText("Right Mouse Button",50.0, 70.0 );
	drawBitmapText("=> Pop-up menu",230.0, 70.0 );
	drawBitmapText("Middle Mouse Button",50.0, 90.0 );
	drawBitmapText("=> Draw the curve",230.0, 90.0 );
	drawBitmapText("F12",50.0, 110.0 );
	drawBitmapText("=> Take screenshots",230.0, 110.0 );
	drawBitmapText("To draw a curve or porcupine: ",50.0, 150.0 );
	drawBitmapText("1. Press Middle Mouse Button",50.0, 170.0 );
	drawBitmapText("2. Click  draw from pop-up menu",50.0, 190.0 );
```

```
        drawBitmapText("3. Choose a method",50.0, 210.0 );
        drawBitmapText("4. Choose corresponding porcupine",50.0, 230.0 );
        drawBitmapText("5. To draw a new curve or porcupine:",50.0, 250.0 );
        drawBitmapText(" First erase the previous curve or porcupine, then draw
again",50.0, 270.0 );

}
```

The pop-up menu and the right- bottom menu are working in synchronization with each other. Values in the sub-menu of Degree of polynomial are the numbers between 0 and N_max. Whenever we choose a number from this sub-menu, the number in the box for the degree of polynomial in the right bottom menu changes accordingly. We provide this by using the “value” flag. Every time we change a number, value becomes equal to that number. The same way, when we change the sub-menu for Porcupines and Curve to Draw or Erase, the corresponding value for the porcupines and curve changes in the right-bottom corner. There is also a current points bar that counts the number of points inserted.







The following code of “DRAW” and “ERASE” box shows a sample for how to design the right-bottom menu. This function takes the color of the box as input with R,G,B values, which is given as white in Display() function.

```
void para_porcupines_draw_erase(float r, float g, float b)
{
	glColor3f(r,g,b);
	glBegin(GL_POLYGON);
		glVertex2i(windowsizeX - 70,windowsizeY - 35 );
		glVertex2i(windowsizeX,windowsizeY - 35 );
		glVertex2i(windowsizeX,windowsizeY);
		glVertex2i(windowsizeX - 70,windowsizeY);
	glEnd();

	glColor3f(0.19, 0.1, 0.21 );
	if (para_porcupines == 1) //when porcupine is on, draw this
		drawBitmapText("DRAW", windowsizeX-65, windowsizeY-10);
	else if (para_porcupines == 0)
		drawBitmapText("ERASE", windowsizeX-65, windowsizeY-10);
	glColor3f(0.19, 0.1, 0.21 );
	glBegin(GL_LINES);
		glVertex2i(windowsizeX,windowsizeY - 35 );
		glVertex2i(windowsizeX,windowsizeY);
		glVertex2i(windowsizeX - 70,windowsizeY);
		glVertex2i(windowsizeX,windowsizeY);
		glVertex2i(windowsizeX - 70,windowsizeY - 35 );
		glVertex2i(windowsizeX,windowsizeY - 35 );
	glEnd();

}
```

In a similar manner, the functions for the other boxes are given in this menus.c source file.

approx.h:

Inside approx.c source file, there is Approximation() function which takes the `X_pts_array and Y_pts_array which` holds the inserted points, number of points N_max, order M, and the coefficient vector of the resulting polynomial X_vector as input and calculates the coefficients of the approximated polynomial :

```
void Approximation(double *x_points, double *y_points, int n_points, int order, double *X_vector)
```

To do the calculation, we use least square approximation method. In this method, for given n points, with f(x) having M (order) unknown coefficients, we build the partial derivatives with respect to the different unknowns and equate them to 0 , to get the minimum value. So, we have a system of equations with M unknowns and M equations. After substituting the equation and taking derivatives, finally we get the following system of equations for 3 unknowns:

$$A\,n + B\sum_{i=1}^{n} x_i + C\sum_{i=1}^{n} x_i^2 = \sum_{i=1}^{n} y_i$$
$$A\sum_{i=1}^{n} x_i + B\sum_{i=1}^{n} x_i^2 + C\sum_{i=1}^{n} x_i^3 = \sum_{i=1}^{n} y_i x_i$$
$$A\sum_{i=1}^{n} x_i^2 + B\sum_{i=1}^{n} x_i^3 + C\sum_{i=1}^{n} x_i^4 = \sum_{i=1}^{n} y_i x_i^2$$

We can write this system in the form of Ax=b , so it becomes:

$$\begin{bmatrix} n & \sum_{i=1}^{n} x_i & \sum_{i=1}^{n} x_i^2 \\ \sum_{i=1}^{n} x_i & \sum_{i=1}^{n} x_i^2 & \sum_{i=1}^{n} x_i^3 \\ \sum_{i=1}^{n} x_i^2 & \sum_{i=1}^{n} x_i^3 & \sum_{i=1}^{n} x_i^4 \end{bmatrix} \cdot \begin{bmatrix} A \\ B \\ C \end{bmatrix} = \begin{bmatrix} \sum_{i=1}^{n} y_i \\ \sum_{i=1}^{n} y_i x_i \\ \sum_{i=1}^{n} y_i x_i^2 \end{bmatrix}$$

The matrix here is represented by A_matrix , [A B C] vector is represented by X_vector, which will give us the solution, and finally the vector on the right hand side is represented by y_vector. We need to allocate spaces for each of these in the dynamic memory, since whenever we change the degree of polynomial in the pop-up menu, the size of A matrix changes.

```
sum_power_x =(double*)malloc(max_power * sizeof(double));
sum_power_yx = (double*)malloc((order)*sizeof(double));
A_matrix = (double*)malloc(order*order*sizeof(double));
b_vector = (double*)malloc(order*sizeof(double));
```

sum_power_yx shows the b vector in Ax=b and according to the given equation we can fill it as follows:

```
for (j = 0; j< order; j++){
      sum_yx = 0.0;
      for (i=0; i<n_points; i++)
      {
            sum_yx += y_points[i]*pow(x_points[i],j);
      }
            sum_power_yx[j]=sum_yx;
      }
b_vector = sum_power_yx;
```

sum_power_x shows the rows of A matrix and we can give sum_power_x and A_matrix as follows:

```
for (j=0; j< order; j++)
	{
	for (k=j ; k< j+order; k++)
		{
		sum_x_pow = 0.0;
		for (i=0; i< n_points; i++)
			{
			sum_x_pow += pow(x_points[i],k);
			}
			sum_power_x[k-j] = sum_x_pow;
			for (l=0; l<order; l++)
			{
				A_matrix[j*order+l] = sum_power_x[l];
			}
		}
	}
```

Then, we use Gauss Elimination method to solve for X_vector.

lagrange.h:

Inside lagrange.c source file, there is Lagrange() function which takes arrays holding inserted x and y points , x_points, y_points, number of points , and an X_value.

```
double Lagrange(double *x_points, double *y_points, int n_points, int order, 
double X_value)
```

This function calculates and returns the corresponding y value for an inserted x value by using Lagrange interpolation method. The point set (x ,y) to do the interpolation is given by the inserted points from the user by clicking the mouse. The following well-known Lagrange algorithm is used to get the polynomial f(x) that connects a given sequence of points:

$$f_L(x) = \sum_{i=0}^{n} L_i(x) f_i \quad where\ L_i(x) = \prod_{\substack{j=0 \\ j \neq i}}^{n} \frac{(x - x_j)}{(x_i - x_j)}$$

Here, L(x) are Lagrange's coefficient polynomial. The following code is used to get the polynomial.

```
	fL=0;
	for (i=0; i< n_points; i++)
	{
		Li= 1;
		for (j=0; j< n_points; j++)
		{
			if (i!=j)
			{
				Li = Li*(X_value - x_points[j])/(x_points[i]-x_points[j]);
			}
		}
		fL = fL+ Li*y_points[i];
	}
	return fL;
```

parametriclagrange.h:

Inside parametriclagrange.c source file, there is Lagrange_Parametric() function which takes arrays holding coordinates of points inserted by the user, x_points, y_points, number of points , a t_value, and a size-2 array to hold the resulting x and y values.

```
void Lagrange_Parametric(double *x_points, double *y_points, int n_points, int
order, double t_value, double pt[2])
```

This function calculates the p(t) values obtained by parametric lagrange interpolation for given t's. The equation is used to do the calculations:

$$p(t) = \sum_{i=0}^{n} L_i(t)Q_i \quad , 0 \leqslant t \leqslant 1 \quad where\ L_i(t) = \prod_{\substack{j=0 \\ j \neq i}}^{n} \frac{(t - t_j)}{(t_i - t_j)} \quad , \quad t_i = \frac{i}{n}$$

A t_vector is used to hold the t values:

```
t_vector = (double*)malloc((n_points)*sizeof(double));
      for (i=0; i<n_points; i++)
      {
            ti=(double)i/(double)(n_points-1);
            t_vector[i]=ti;
      }
```

The following code shows the algorithm:

```
sum1=0.0; sum2 =0.0;
for (i=0; i< n_points; i++){
      product= 1;
      for (j=0; j< n_points; j++){
            if (i!=j){
            product =product*(t_value - t_vector[j])/(t_vector[i]-t_vector[j]);
                        } }
            Li= product;
            sum1= sum1 + Li*x_points[i];
            sum2= sum2 + Li*y_points[i];
}
            pt[0] = sum1;
            pt[1] = sum2; }
```

The results x(t) and y(t) values are held by the pt array, where pt[0] is x(t) and pt[1] is y(t).

porcupine.h:

Inside porcupine.c source file, there is Porcupine() function which takes arrays holding coordinates of points on the polynomial curve , curveXarr, curveYarr, number of points on the curve curvepoints, windowsize, arrays of coordinates of porcupine points porcupine_x, porcupine_y.

```
void Porcupine(double *x_points, double *y_points, int n_points, int window_X,
double *porc_x, double *porc_y )
```

This function calculates the porcupine curves of the obtained polynomial by calculating the curvature of the oscillating circle at each interval point. To do this, we take triples of curve points, and by considering the third one as origin and removing the others accordingly, we find the center and radius of the oscillating circle at each point. Before that, we need an RUNIT value, here we set it as windowsizeX/20. After we find the radius, curvature k is equal to $(RUNIT^2)/r$. The distance from the center of the circle is the same with the length of the curvature k. So the ratio of x coordinate and y coordinates of the center (SX/SY) is equal to that of the x coordinate and y coordinates of curvature point (DX/DY) are equal. We set this ratio to F. Finally, $DY = k / (sqrt(1+F^2))$. However, we need to consider the sign of the curvature. So, we define a SIGN function to get the signed distance of the porcupine point. Also, we need to set the conditions that when determinant is equal to 0, curvature is zero and when SY = 0, DY = 0 and curvature is equal to DX. Finally , we need to move our point back to its original place and then add DX to x coordinate of the curve and subtract DY from the y coordinate. The following code represents this algorithm.

```
for (i = 0; i< n_points-2; i++)
            {
                  x01=  x_points[i] - x_points[i+2];
                  y01=  y_points[i] - y_points[i+2];

                  x02=  x_points[i+1] -x_points[i+2];
                  y02=  y_points[i+1] - y_points[i+2];

                  d = (y02 * x01) - (y01 * x02);

                  z01 = -((x01*x01)+(y01*y01));
                  z02 = -((x02*x02)+(y02*y02));

                  a = (y02*z01-y01*z02)/(y02*x01-y01*x02);
                  b = (z02*x01-z01*x02)/(y02*x01-y01*x02);

                  //center of the circle with translated coordinates
                  CX = -(a/2);
                  CY = -(b/2);

                  //radius of the circle
                  r = sqrt(CX*CX+CY*CY);

                  SX= x02 - CX;
                  SY = CY-y02;

                  k = pow(RUNIT,2) / r;

                  if (d == 0) { k=0; DX = 0; DY = 0; }
                  else if ( SY == 0) {
                        DY = 0;
                        DX = k;
                        DX = SIGN(DX, SX);}
                  else {
                        F = (SX)/ (SY);
                        DY = k/ sqrt(1+pow(F,2));
```

```
                    DY = SIGN(DY,SY);
                    DX = DY * F;
                    }
              porc_x [i] = (x02 + x_points[i+2] + DX );
              porc_y [i]= (y02  + y_points[i+2] - DY);
        }
```

draw.h:

Inside draw.c source file, we have functions for drawing the polynomial curves and porcupine curves. First, we allocate spaces for curve points, and porcupine points on the polynomial curve. Also we allocate new arrays for the points of curves and porcupines of the parametric lagrange and parabolic blending since we need a bigger sized arrays for those curves. The number of porcupine points is set as a ratio of window size , `windowsizeX / 5 +1` . The number of curve points is 2 more than this value since to calculate porcupines we need three curve points for each porcupine and the first and the last curve points can not be used for further calculations. The number of parametric porcupine points is set as 3 times the number of nonparametric curvepoints. The number of parametric curve points is again 2 more than parametric porcupine points. `CurveXarr and curveYarr` hold the x and y coordinates of the curve points. `porcupine_x and porcupine_y h`old the coordinates of the porcupine points. `CurveXarr and curveYarr` hold the x and y coordinates of the curve points. `porcupine_x and porcupine_y h`old the coordinates of the porcupine points. `curveXpara and curveYpara` hold the x and y coordinates of the parametric curve points. `para_porc_x and para_porc_y h`old the coordinates of the parametric porcupine points.

For Lagrange and Approximation methods, we need to put the inserted points in order, since they are not parameterized, there's no order in them. To get the maximum and minimum value of these points, we use maxmin() function.

To draw the curve of lagrange and approximation methods, first we get a stepsize dx to move from the minimum point till the end in the x direction. To do that, we calculate dx by taking the subtraction of max and min points and dividing them by the number of intervals, which is 1 less than the number of curve points. To find the next x coordinate on the curve, we add this dx to the current x value.

For Approximation method, to find the corresponding y value, we call our Approximation function inside approximation.h from main.c. Then, we call draw_curve() function to get the figure. In draw_curve(), we calculate equation of the polynomial by adding the product of elements of X_vector and the powers of corresponding x value. The following code shows this:

```
void draw_curve(void)
{
      minmax(X_pts_array, N_max);
      dx = (MAX - MIN)/(curvepoints -1);

      glColor3f(0.19, 0.1, 0.21 );
      glBegin(GL_LINE_STRIP);
      for(i=0; i<curvepoints; i++)
      {
            curveX = MIN + dx*(double)(i);
            curveXarr[i]=curveX;
            curveY = X_vector[0];
```

```
            for (j=1; j<M; j++)
            {
                  curveY+= X_vector[j]*pow(curveX, j);
            }
            curveYarr[i] = curveY;

            glVertex2d(curveX, curveY);
      }
      glEnd();
}
```

For Lagrange polynomial, there's a draw_lagrange() function that calls our Lagrange() function from lagrange.h and equates the result to y. Then draws the figure according to the y values and given x values. The following code shows this:

```
void draw_lagrange(void)
{
      dx = (MAX - MIN)/(curvepoints -1);
      glColor3f(0.19, 0.1, 0.21 );
      glBegin(GL_LINE_STRIP);
      for(i=0; i<curvepoints; i++)
      {
            curveX = MIN + dx*(double)(i);
            curveXarr[i]=curveX;
            curveYarr[i] = Lagrange(X_pts_array, Y_pts_array, N_max, M,curveX );
            curveY = curveYarr[i];

            glVertex2d(curveX, curveY);
      }
      glEnd();
}
```

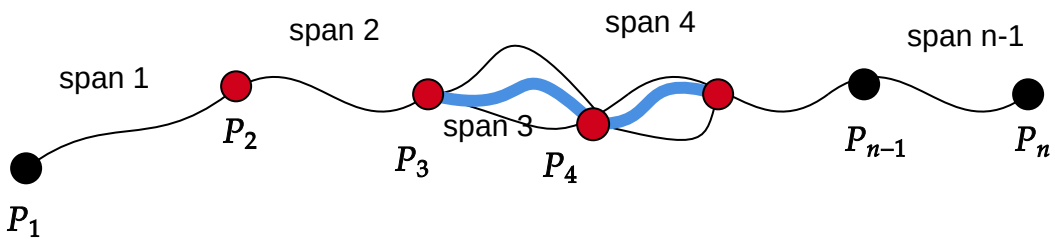
For Parametric Lagrange method, we have a draw_parametric_lagrange() function that calls the Lagrange_Parametric() function of the parametriclagrange.h file to find the result p(t). First, t that is going to sent into the Lagrange_Parametric() function is assigned to 0. Then, a stepsize dt is calculated. This dt is chosen to be 1 / (para_points-1) since t has a range from 0 1 and the number of allocated points for parametric functions is para_points. The values of pt[] array, which will get the x(t) and y(t) values to draw our figure, is assigned to 0. A k value, which will count the number of points on the parametric curve, is decleared and assigned to 0. `for (t = 0.0 ; t<=1.0+ dt; t += dt)` loop is used to calculate parametric lagrange method and draw the figure. curveX and curveY , which are going to draw our figure , are assigned to the obtained pt[0] and pt[1] values. The elements of curveXpara and curveYpara arrays, that will finally draw the parametric porcupines, are assigned to these curveX and curveY values. The k is increased by 1 for each loop. The following code illustrates this:

```
void draw_parametric_lagrange(void)
{     t=0.0;
      dt=(double)(1.0)/(double)(para_points-1);
      double pt[2] = {0.0,0.0};
      glColor3f(0.19, 0.1, 0.21 );
      glBegin(GL_LINE_STRIP);
```

```
	int k;
	k=0;
		for (t = 0.0 ; t<=1.0+ dt;  t += dt){
		Lagrange_Parametric(X_pts_array, Y_pts_array, N_max, M,t, pt);
		curveX = pt[0];
		curveY = pt[1];
		curveXpara[k]=curveX;
		curveYpara[k]=curveY;
		k++;
		glVertex2d(curveX, curveY);
		}
	glEnd();
}
```

To draw the a parabolic blending curve, the below improved parabolic blending algorithm is used. For this, a new source file or function is not created since it would be hard to draw each interval by calling a function. Therefore, the calculations and drawings are done inside draw_parabolic_blending() function.

For improved parabolic blending method there are n-1 spans that connect n given points. For the first span, to get the blending function 3 points are used. For the spans starting from 2 to n-2, groups of 4 points are used for blending and for the last span, the last 3 points are used. The blending functions and algorithm are given in the following:



In the above figure, for span 3 the red dots are used to get the blending functions. The blue curve between P3 and P4 represents the blending curve for span 3. The next blue curve is that for the next span. This way, the blending curves are connected and a new curve is obtained.

given n points P_i, $i = 0, 1, 2, \ldots, n$ $(n \geq 3)$

for span 1 (*first span*):

$$Q(t) = P_1 \times LP_1 + P_2 \times LP_2 + P_3 \times LP_3 \ , \ 0 \leq t \leq 1 \ , \ \textit{where}$$

$$LP_1 = \frac{1}{2}(t-1)(t-2)$$
$$LP_2 = -t(t-2)$$
$$LP_3 = \frac{1}{2}t(t-1)$$

for span k, $2 \leqslant k \leqslant n-2$:

$$Q(t) = P_{k-1} \times A_0 + P_k \times A_1 + P_{k+1} \times A_2 + P_{k+2} \times A_3 \,,\ 1 \leqslant t \leqslant 2,\ \textit{where}$$

$$A_0 = -\frac{1}{2}(t-1)(t-2)^2$$

$$A_1 = \frac{1}{2}(t-1)(t-2)(t-3) + t(t-2)^2$$

$$A_2 = -\frac{1}{2}\, t\,(t-1)(t-2) - (t-1)^2(t-3)$$

$$A_3 = \frac{1}{2}(t-1)^2(t-2)$$

for span $n-1$ (last span) :

$$Q(t) = P_{n-2} \times LQ_1 + P_{n-1}LQ_2 + P_nLQ_3 \,,\ 2 \leqslant t \leqslant 3 \,,\ \textit{where}$$

$$LQ_1 = \frac{1}{2}(t-2)(t-3)$$

$$LQ_2 = -(t-1)(t-3)$$

$$LQ_3 = \frac{1}{2}(t-1)(t-2)$$

Here the dt is determined to be 3 / (curvepoints-1) . Since t starts from 0 and goes up to 3. A u variable, that will count the curveXpara and curveYpara array elements is assigned to 0. These arrays are going to be used for drawing porcupines. Then the for loop `for (t = 0.0 ; t<=1.0;  t += dt)` is started that will calculate the values for the first span. The little line segments for the blending functions are drawn within the loop. This is given in the following code:

```
dt=(double)(3.0)/(double)(curvepoints-1);
glColor3f(0.19, 0.1, 0.21 );
glBegin(GL_LINE_STRIP);
      // span 1, first 3 points
int u;
t=0.0;
u=0;
for (t = 0.0 ; t<=1.0;  t += dt){
      LP0 = ((double)(1)/(double)(2)) * (t -1)*(t - 2);
      LP1 = (-t) * (t-2);
      LP2 = (t) * (t-1) / (double) 2;
      qt0 = X_pts_array[0] * LP0 + X_pts_array[1] * LP1+ X_pts_array[2] * LP2;
      qt1 = Y_pts_array[0] * LP0 + Y_pts_array[1] * LP1 + Y_pts_array[2] * LP2;
      curveX = qt0;
      curveY = qt1;
```

```
      curveXpara[u]=curveX;
      curveYpara[u]=curveY;
      u++;
      glVertex2d(curveX, curveY);}
```

For the middle spans, two loops are used, one for the points k from 2 to n-2 and another for t's, which goes from 1 to 2. For each loop, the corresponding values for curve points are obtained and the little lines are drawn, u is increased by 1 and given into curveXpara and curveYpara array elements to get the to draw porcupines. The following code illustrates this:

```
int k;
for (k= 1; k<N_max-2; k++){
      for (t=1.0; t<=2.0; t+=dt){
            A0 = ((double)(-1)/(double)(2))*(t-1) *(t-2)*(t-2);
            A1 = ((double)(1)/(double)(2))*(t-1)*(t-2)*(t-3)+ t*(t-2)*(t-2);
            A2 = ((double)(-1)/(double)(2))*(t)*(t-1)*(t-2)- (t-1)*(t-1)*(t-3);
            A3 = ((double)(1)/(double)(2))*(t-1)*(t-1)*(t-2);
            qt0 = X_pts_array[k-1] * A0 + X_pts_array[k] * A1 +
X_pts_array[k+1]*A2 + X_pts_array[k+2] * A3;
            qt1 = Y_pts_array[k-1] * A0 + Y_pts_array[k] * A1 +
Y_pts_array[k+1]*A2 + Y_pts_array[k+2] * A3;
            curveX = qt0;
            curveY = qt1;
            curveXpara[u]=curveX;
            curveYpara[u]=curveY;
            u++;
            glVertex2d(curveX, curveY);}}
```

For the last span, t goes from 2 to 3. Similar procedures and the above algorithm is used to get the curve points and draw the line segments and thus curves. Since these curves needed to be connected to each other, the glEnd() function is added after the last loop.

```
//last 3 points
for (t = 2.0 ; t<=3.0;  t += dt) {
      LQ1 = (t - 2) * (t - 3) /(double)2 ;
      LQ2 = (double)(-1)*(t - 1) * (t - 3);
      LQ3 = ((t - 1)*(t - 2)) /(double)2 ;
      qt0 = X_pts_array[N_max - 3] * LQ1 + X_pts_array[N_max -2] * LQ2 +
X_pts_array [N_max-1] * LQ3;
      qt1 = Y_pts_array[N_max - 3] * LQ1 + Y_pts_array[N_max -2] * LQ2 +
Y_pts_array [N_max-1] * LQ3;
      curveX = qt0;
      curveY = qt1;
      curveXpara[u]=curveX;
      curveYpara[u]=curveY;
      u++;
      glVertex2d(curveX, curveY);
      }
      glEnd();}
```

To draw the porcupines, we draw lines from each curve point starting from index 1 to corresponding porcupine point, which starts from 0 index of porcupine points array. There are two porcupine

functions, draw_porcupine() and draw_parametric_porcupine() , one for explicit curves, another for parametric curves. These functions can be called after the Porcupine() function call to draw the porcupines.

For explicit porcupines the following code is used, curveXarr and curveYarr shows the curve points array and porcupine_x and porcupine_y shows the porcupine points array. Corresponding curve and porcupine points are connected to get the porcupine curves.

```
void draw_porcupine(void)
{
	glColor3f(0.0f, 0.0f, 1.0f);
	glBegin(GL_LINES);
	for(i = 1; i < porc_vertices+1; i++)
	{
		curveX = curveXarr[i]; // 1,2,..., porc_vertices
		curveY = curveYarr[i];
		porcX = porcupine_x[i-1]; // 0,1,...porc_vertices-1
		porcY = porcupine_y[i-1];
		glVertex2d(curveX, curveY);
		glVertex2d(porcX,porcY);
	}
	glEnd();
}
```

For parametric porcupines the following code is used, curveXpara and curveYpara shows the curve points array and para_porc_x and para_porc_y shows the porcupine points array. New and bigger arrays are allocated for parametric porcupines since we need more points than explicit porcupines to get a better looking curve. Corresponding curve and porcupine points are connected to get the porcupine curves.

```
void draw_parametric_porcupine(void){
	glColor3f(0.0f, 0.0f, 1.0f);
	glBegin(GL_LINES);
	int v;
	for(v = 1; v < para_porc+1; v++){
		curveX = curveXpara[v]; // 1,2,..., porc_vertices
		curveY = curveYpara[v];
		porcX = para_porc_x[v-1]; // 0,1,...porc_vertices-1
		porcY = para_porc_y[v-1];
		glVertex2d(curveX, curveY);
		glVertex2d(porcX,porcY);}
		glEnd(); }
```

screenshot.h:

Inside screenshot.c file, there's a function to take screenshot of the screen when F12 is clicked.

mouse.h:

Inside mouse.c , we define the functions for mouse movement.

main.c:

Inside main.c file, there are main() and Display() functions, where everything starts , the inputs are taken, the functions are called and the flags are assigned here. Some of the right bottom menu elements are assigned here.

Following flags are used to draw and erase distinct curves:

```
if (value == N_max+2  ) para_porcupines = 1;

else if (value == N_max+3) para_porcupines =0;


if (value == N_max+9 )  curve_on = 1;
else if (value == N_max+10) curve_on = 0;


if (value == N_max+6) lagrange_on =1;
if (value == N_max+7) para_lagrange_on =1;
if (value == N_max+8) blending_on =1;

if (value == N_max )
      {
      porcupines = 1;
      }
else if (value == N_max + 1)
      {
      porcupines = 0;
      }
```

Here, the values are the corresponding values given inside the pop-up menu. When we click the buttons the flags take into action. I had difficulties using the pop-up menu in a proper way. Whenever I click on the draw button, it drew and when I click again it erased the figure, leaving just the points. After using the flags this problem is solved. But then it started drawing one on to another curve. To prevent this, after many trials, I asked the user within the left top instructions to erase the curve and porcupine first with the erase buttons and then draw again. That way, it draws the figures and porcupines properly , however, there might be a better way to do this.

The following right bottom menu elements are assigned here, maximum number of points, `N_max_char`, degree of polynomial, `M_char`, current number of points, `npoints_char` and methods used.

```
sprintf(N_max_char, "%d", N_max);
sprintf(M_char,"%d", M-1 );
sprintf(npoints_char,"%d", N_points);

if (value > 0 && value < N_max) {
      M = value + 1;
      approximation_on =1;
      X_vector = (double *)realloc(X_vector, sizeof(double)*M);
      sprintf(M_char, "%d", M-1);
      sprintf(methods, "APPROXIMATION");}

if (value == N_max+6 ){
      sprintf(methods, "LAGRANGE INTERPOLATION");
      sprintf(M_char,"%d", N_max-1 );}
```

```
      if (value == N_max+7){
            sprintf(methods, "PARAMETRIC LAGRANGE");
            sprintf(M_char,"%d", N_max-1 );}

      if (value == N_max+8){
            sprintf(methods, "PARABOLIC BLENDING");
            sprintf(M_char,"%d", N_max-1 );}

      if (value == N_max+5){
            sprintf(methods, "METHOD");
            sprintf(M_char,"%d", N_max-1 );}
```

The values here again are the ones from pop-up menu.

Following function calls are used to draw the right bottom and left top instructions menus.

```
      instructions();
      method(1.0,1.0,1.0);
      number_of_points(1.0,1.0,1.0);
      N_max_input(1.0,1.0,1.0);
      current_number(1.0,1.0,1.0);
      current_number_input(1.0,1.0,1.0);
      degree_of_polynomial(1.0,1.0,1.0);
      polynomial_input(1.0,1.0,1.0);
      menu_curve(1.0,1.0,1.0);
      curve_on_off(1.0,1.0,1.0);
      menu_porcupines(1.0,1.0,1.0);
      porcupines_draw_erase(1.0,1.0,1.0);
      menu_parametric_porcupines(1.0,1.0,1.0);
      para_porcupines_draw_erase(1.0,1.0,1.0);
```

To draw the curves and porcupines, the flags are used in the following way, closed =1 occurs with the middle button-click .

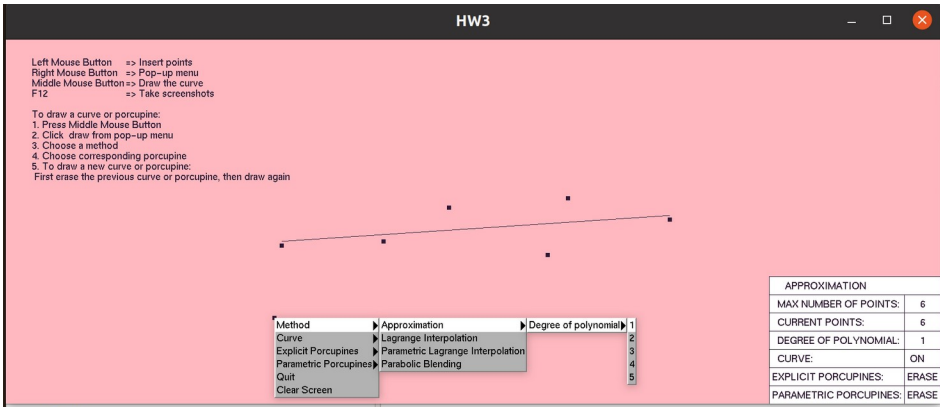
```
curve_and_porcupine_points();

if (curve_on == 0) {
      approximation_on =0;
      lagrange_on = 0;
      para_lagrange_on =0;
      blending_on=0;}

if (closed == 1 && curve_on ==1) {
      if (approximation_on == 1 ){
            Approximation(X_pts_array, Y_pts_array, N_max, M, X_vector);
            draw_curve();}

      if (lagrange_on == 1 ){ draw_lagrange();}
      if (para_lagrange_on==1){ draw_parametric_lagrange();}
      if (blending_on ==1){draw_parabolic_blending();} }

      if (closed == 1 && porcupines == 1) {
            Porcupine(curveXarr, curveYarr, curvepoints, windowsizeX, porcupine_x,
porcupine_y);
            draw_porcupine();}
```

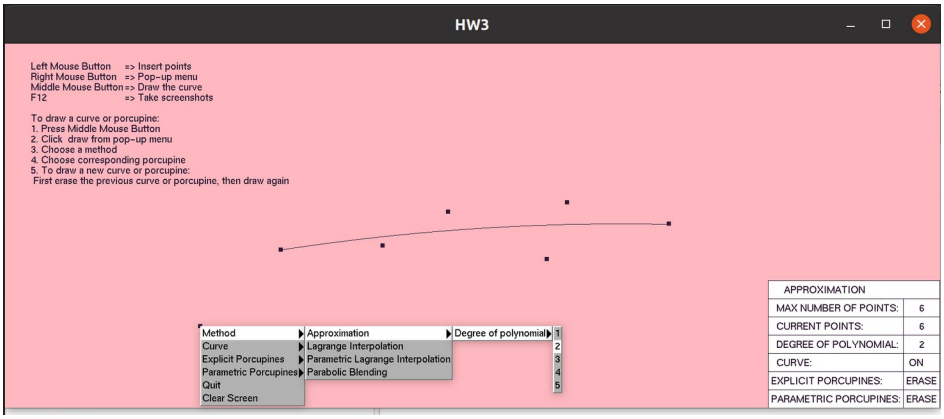
```
      if (closed == 1 && para_porcupines == 1 ) {
            Porcupine(curveXpara, curveYpara, para_points, windowsizeX,
para_porc_x, para_porc_y);
            draw_parametric_porcupine();}
```

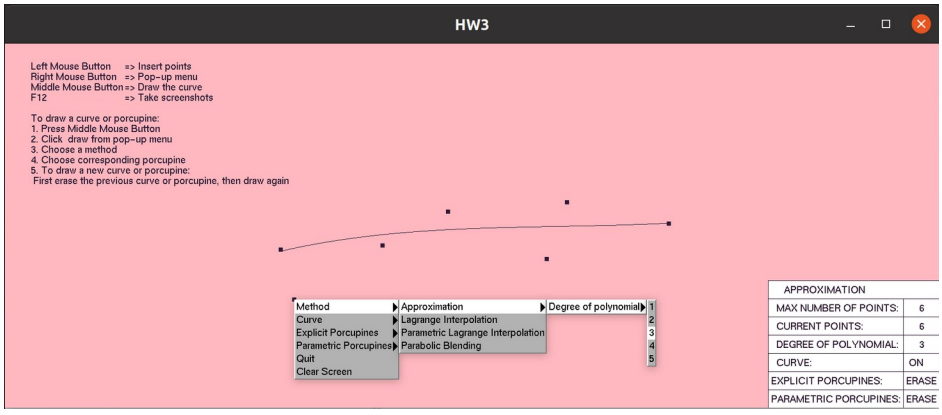
In the end of main function, all the allocated arrays are freed.

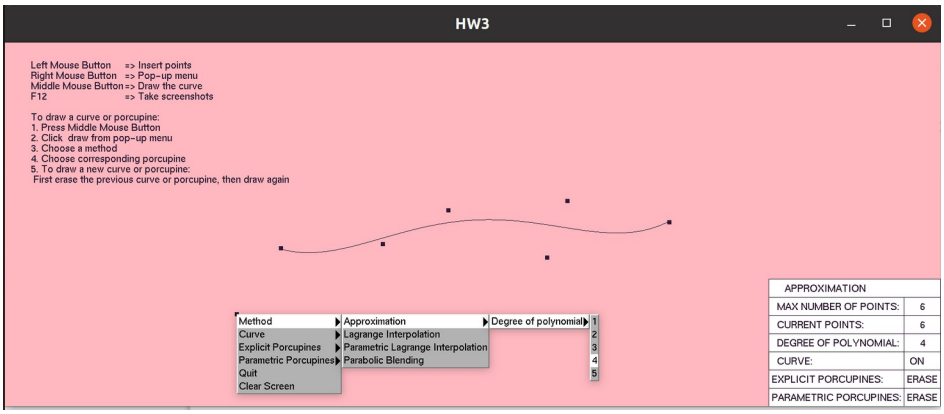
Sample Figures:

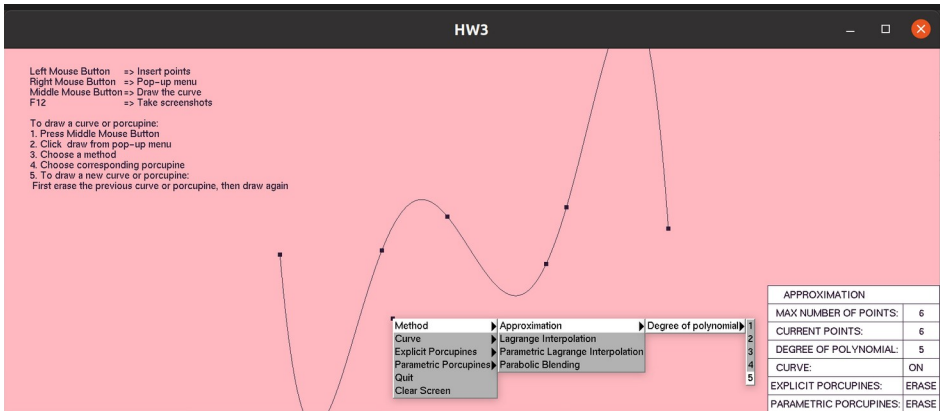
In following figures, we can see polynomial curves with approximation method starting from degree 1 to 5 with inserted 6 points:



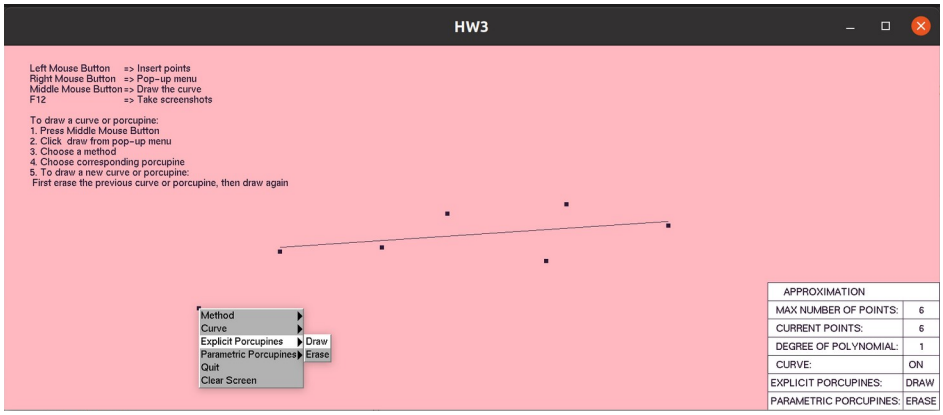


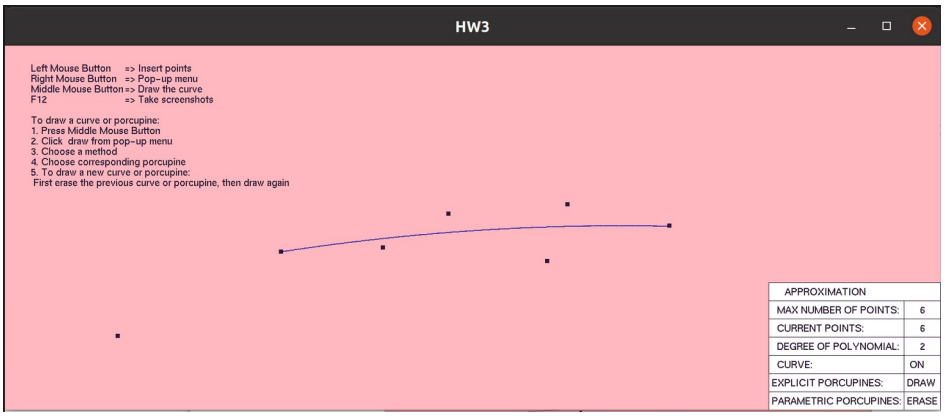


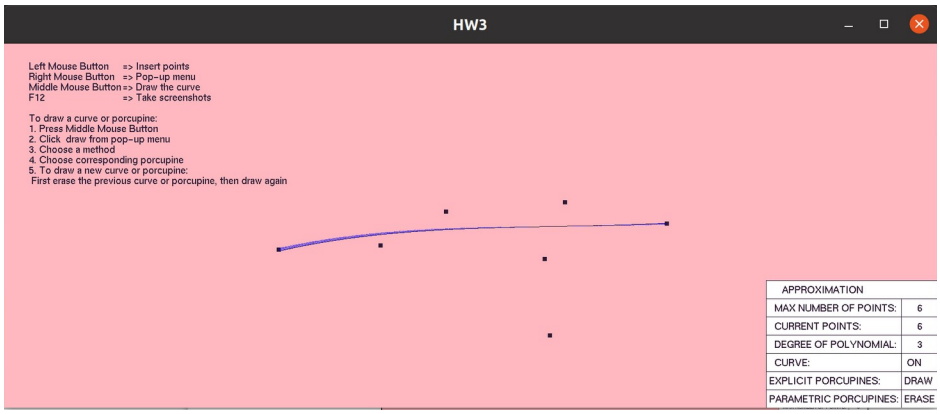


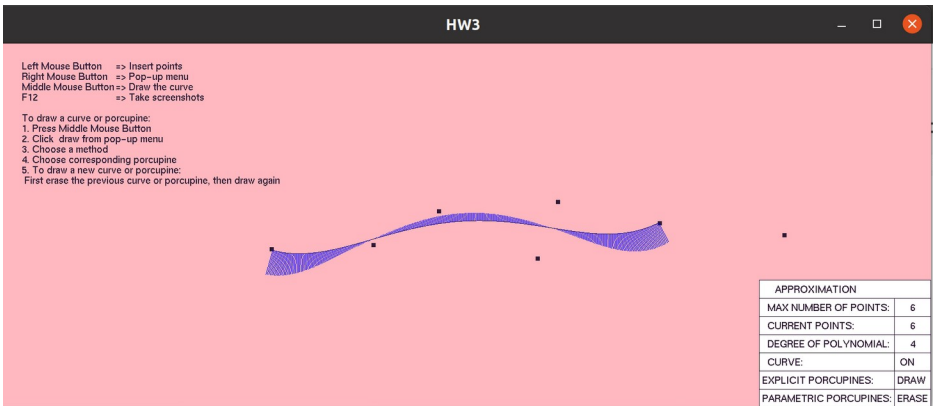


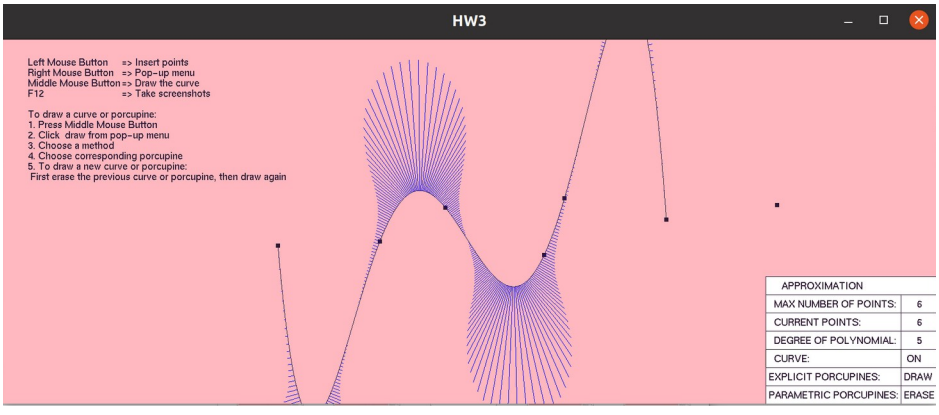
The following figures shows porcupine curves for each curve above, Explicit porcupines are set to draw mode here:



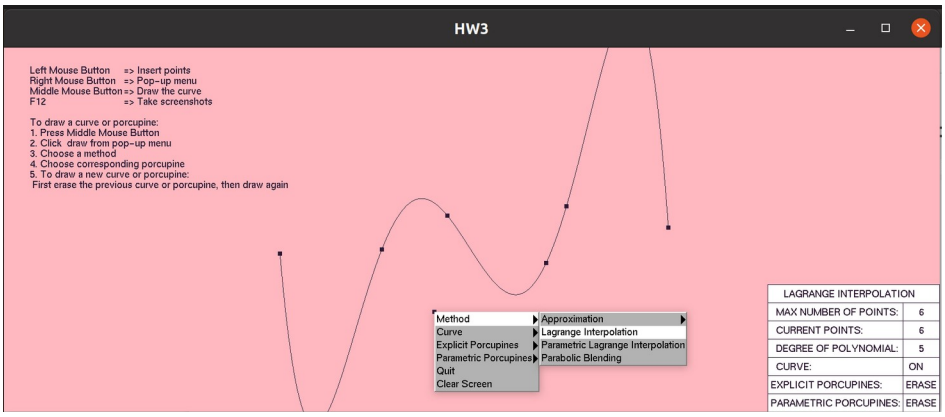

HW3
Left Mouse Button => Insert points
Right Mouse Button => Pop-up menu
Middle Mouse Button => Draw the curve
F12 => Take screenshots
To draw a curve or porcupine:
1. Press Middle Mouse Button
2. Click draw from pop-up menu
3. Choose a method
4. Choose corresponding porcupine
5. To draw a new curve or porcupine:
First erase the previous curve or porcupine, then draw again
APPROXIMATION
MAX NUMBER OF POINTS: 6
CURRENT POINTS: 6
DEGREE OF POLYNOMIAL: 2
CURVE: ON
EXPLICIT PORCUPINES: DRAW
PARAMETRIC PORCUPINES: ERASE


HW3
Left Mouse Button => Insert points
Right Mouse Button => Pop-up menu
Middle Mouse Button => Draw the curve
F12 => Take screenshots
To draw a curve or porcupine:
1. Press Middle Mouse Button
2. Click draw from pop-up menu
3. Choose a method
4. Choose corresponding porcupine
5. To draw a new curve or porcupine:
First erase the previous curve or porcupine, then draw again
APPROXIMATION
MAX NUMBER OF POINTS: 6
CURRENT POINTS: 6
DEGREE OF POLYNOMIAL: 3
CURVE: ON
EXPLICIT PORCUPINES: DRAW
PARAMETRIC PORCUPINES: ERASE

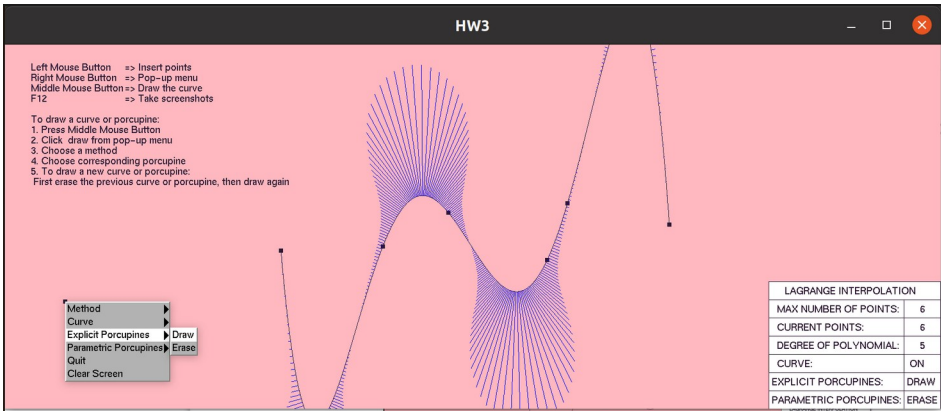

HW3
Left Mouse Button => Insert points
Right Mouse Button => Pop-up menu
Middle Mouse Button => Draw the curve
F12 => Take screenshots
To draw a curve or porcupine:
1. Press Middle Mouse Button
2. Click draw from pop-up menu
3. Choose a method
4. Choose corresponding porcupine
5. To draw a new curve or porcupine:
First erase the previous curve or porcupine, then draw again
APPROXIMATION
MAX NUMBER OF POINTS: 6
CURRENT POINTS: 6
DEGREE OF POLYNOMIAL: 4
CURVE: ON
EXPLICIT PORCUPINES: DRAW
PARAMETRIC PORCUPINES: ERASE


HW3
Left Mouse Button => Insert points
Right Mouse Button => Pop-up menu
Middle Mouse Button => Draw the curve
F12 => Take screenshots
To draw a curve or porcupine:
1. Press Middle Mouse Button
2. Click draw from pop-up menu
3. Choose a method
4. Choose corresponding porcupine
5. To draw a new curve or porcupine:
First erase the previous curve or porcupine, then draw again
APPROXIMATION
MAX NUMBER OF POINTS: 6
CURRENT POINTS: 6
DEGREE OF POLYNOMIAL: 5
CURVE: ON
EXPLICIT PORCUPINES: DRAW
PARAMETRIC PORCUPINES: ERASE

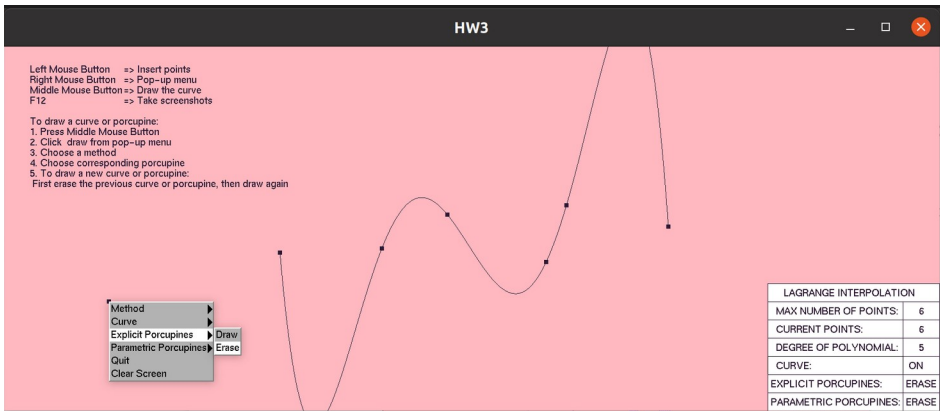
In the following figure, the figure is drawn using Lagrange interpolation method.



The following figure shows the porcupines for the above figure, explicit porcupines are set to curve on mode here.

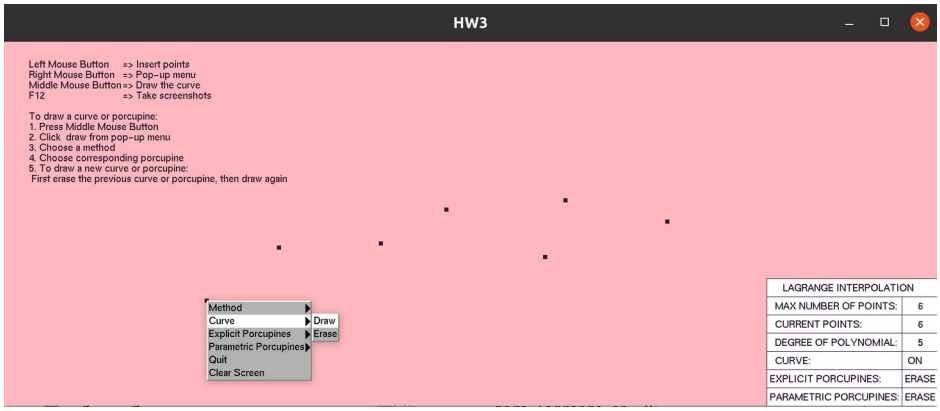


We can see that approximation method with 5 degree polynomial gives the same figure with Lagrange interpolation. We use erase button from Explicit Porcupines menu to erase the porcupines.

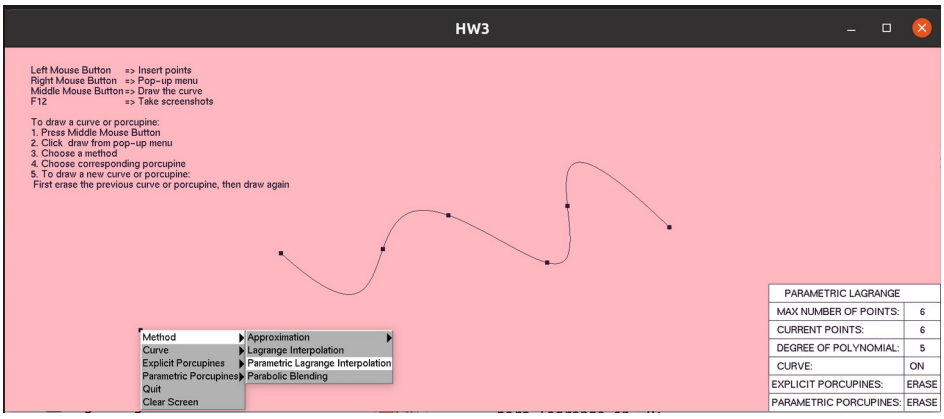


We use erase button from Curve menu to erase the curve. The right bottom menu becomes OFF, when we do this.

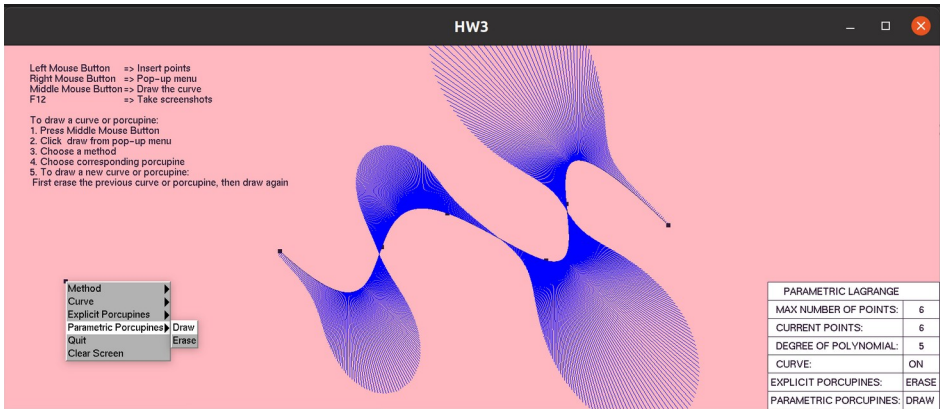
To draw a new figure we set the Curve to draw mode.



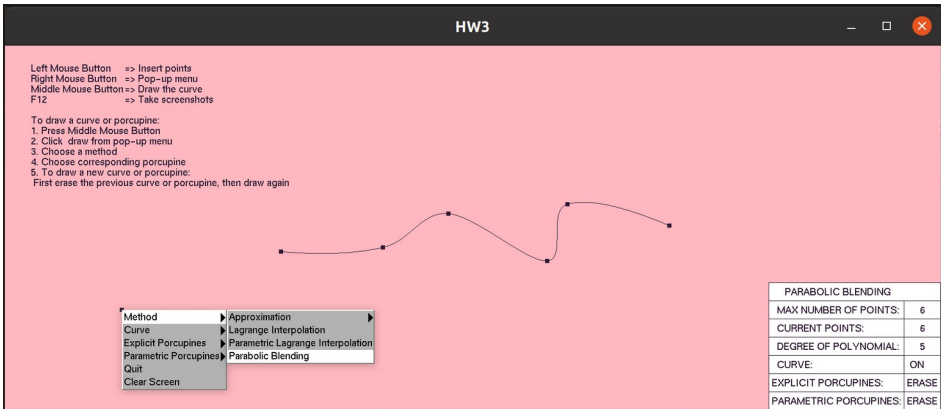
Then, choose the method from the Method menu. This time we use Parametric Lagrange Method.



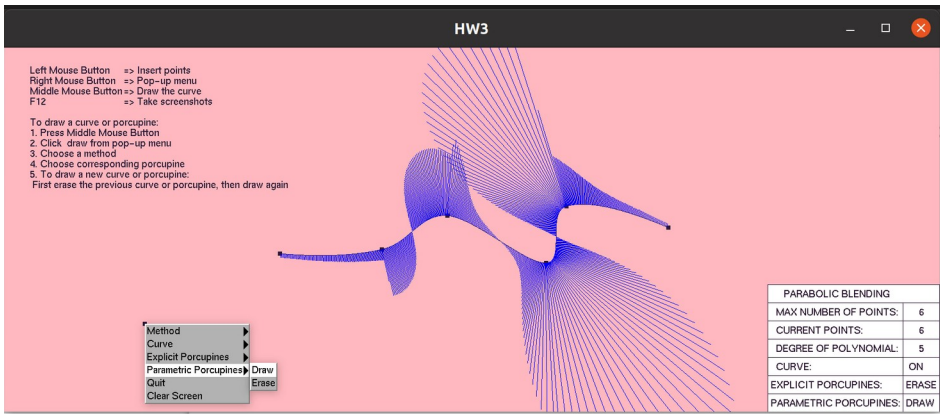
In the below figure, we see the porcupines for this figure. The porcupines are drawn by Parametric Porcupines menu.



The following figure is drawn by the Parabolic Blending method.



The following figure shows the porcupines for this figure. We use parametric porcupines to draw the figure. We can see that at the points there is C1 continuity and between two points where the blending curves occur, there's C2 continuity.



The clear screen button helps to clear the screen back to start and Quit button helps to close the window and quit. F12 key takes screenshots.

Observations:

This OpenGl graphics program includes for different methods to draw polynomial from given points. To write the algorithms and connect those with the graphics program was a hard task to do since there are more than one tasks. Lagrange Interpolation and Approximation were easier compared to the other methods since there are already x and y values given from the user. For parametric methods, converting t values back to x and y values and making them draw took most of my time. Then, I realized that a simple variable mistake caused all the trouble. I tried using different loops, loops for curve points. At last, decided to use t in the loop and use the curve points index as a counter. That made the work load easier. That took weeks to decide on.

For parabolic blending, first I tried to put the algorithm into a function within a new source file and tried to call from draw source file. However, that didn't work. Since there needed to be line segments between and within each span, holding them into an array and calling them from outside was impossible for me. Then I decided to use it within the draw source file, without calling any other function. All the little line segments are calculated and drawn here. Again, here t is used as a loop and the curve points array indices are used as counters. I tried the opposite way around, making an indices loop and incrementing t as a counter, however, that didn't work. Then, this way it worked better. After managing to draw the curve, I had difficulties drawing the porcupines. This curve array indices way helped to hold the points for drawing porcupines. However, for the parabolic blending method, when I tried more points to insert, the porcupines were incomplete, some of them were missing, although they were there for smaller numbers. Then I realized that the number of curve points was not enough to draw the porcupines for parabolic blending. To do this, I declared a new variable for holding the number of curve points for parametric methods, and decided to make the value to be `windowsizeX / 3 +1 ,`however, that didn't work. Since the curvepoints are counted with a different measure, which was divided by 5, the program drew the porcupines far from the curve. Then, I decided to keep the curvepoints as it is but change this new variable to be a power of curvepoints. That way, it worked and drew the porcupines even for 10,11, 12 points properly. Here, at last I chose the number of parametric porcupine points to be 3 times the porcupine points. Then, the parametric curve points will be 2 more than this value.

Another issue that I had struggle with was the right click pop up menu buttons use. It was hard to manage those. First, I decided to put the conditions according to values but I realized that this was a big mistake. Since value is changing all the time when we click on the other buttons, the buttons couldn't do their job properly. To solve this issue, I used flags for each button instead of value, this way it worked. However, there's still a problem that they draw onto each other, if we keep clicking on the Methods or the porcupines. I tried different ways to prevent this, like closing the Lagrange flag when drawing Approximation for example, but this doesn't work since it remains closed, and if we want to draw it again, it won't draw. Then, I deleted this and kept it as it is and asked the user to delete the figure or porcupines first, then draw a new one. This way, if we follow the instructions, everything works fine. Yet, I think, there might be a better way to do this. These are some of my observations from

this task. I had a lot of struggles during this journey, I am glad to solve most of them. There remains some, which wait to be solved in future.

Conclusion:

In conclusion, here a program that asks from the user to insert number of points from console, and to draw points by left clicking the mouse and draws the selected polynomial curve and porcupine curves from these points is explained. An interactive graphics program that performs k th order polynomial approximation and creates porcupines for each approximated polynomial is done here. It also interpolates the given points using both parametric Lagrange Polynomial Method and Improved Parabolic Blending Method. To do this program C language is used on an Eclipse platform with OpenGl graphics tool. Program draws “Lagrange polynomial”, “Parabolic Blending”, “Parametric Lagrange Interpolation” and “Approximation” curves for the corresponding points by clicking the corresponding buttons. User draws parametric porcupines and explicit porcupines by clicking the corresponding buttons for the corresponding curves. User can erase the corresponding porcupines drawn on the curve without erasing the curve. In order to draw a new curve, the older curve must be erased by clicking the “erase” button under the curve button. “Polynomial degree” is displayed within the right bottom corner menu. The user can change the degree of polynomial for Approximation by clicking the degree choices from the pop-up menu. Program can get a screenshot output in the tga format by clicking pressing the F12 key. User can clear the screen using clear screen button from the pop-up menu and can exit by clicking the “Quit” button.