Since we are starting at the very beginning, our first step will be to define the problem that the techniques of computer graphics attempt to solve. So to begin our discussion, we shall make a first (clumsy) attempt at explaining the problem by saying that the main problem in computer graphics is how to create a desired image.

Given the number of different reasons people want to create images, a statement that encompasses all of computer graphics must necessarily be vague. In order to give a better sense of what we are trying to understand, we’ll discuss some specific problems that computer graphics attempts to solve.

One of the most common uses for computer graphics is the creation of the image that looks like a photograph from real life, but portrays something we could not actually take a picture of. Foe example: we might want to create an animation for a movie that portrays an alien creation or location, or we might want to create an architectural rendering to get an idea of what the final building will look like. We’ll call this type of computer graphics photo-realistic. Typically there is a significant amount of time available for creating these images, and so the technique often attempt to capture the properties of lights and surfaces as accurately as possible, at the expense of time needed to compute the image.

Another common use for computer graphics is the creation of the images very quickly so that they can be used in an interactive animation; the application of these algorithms that the most people are familiar with is computer games. As a user controls, their character, we want to creates a series of images very quickly (hopefully 30 per seconds or more) that respond to their inputs. We’ll refer to this type of computer graphics as real-time or interactive. We might want these images to look as realistic as possible, but the fact of the situation is that if we have more time to devote to computation, we can create better results. Thus, real-time computer graphics is usually concerned with the techniques of creating the best looking picture possible with the very short amount of time available between frames.

In the text that follows, we’ll begin by attempting to understand the concepts of pixels and how computer represent color, which is the format for what computer graphics algorithm output. After that, we’ll consider how we represent the input data that programs in computer graphics use, the 3D objects that get rendered. With these basics out of the way, we’ll then delve into the techniques of photo-realistic and real-time rendering.

**The Borland Graphics Interface (BGI) library of functions uses the following co-ordinate system:**

Top-Left of Window : ( 0 , 0 )

Top-Right of Window : ( max X , 0 )

Bottom-Left of Window : ( 0 , max Y )

Bottom-Right of Window : ( max X , max Y )

In normal VGA mode, max X is 1279 and max Y is 719 which provides for a (1280 X 720) screen resolution. We must also specify the following “#include” statements when using graphics in a program:

#include<stdlib.h>

#include<graphics.h>

The following is the code which can be used as a graphics setup program when we need to begin coding a new graphics program. The purpose of the code is to initialize the graphics functions and make sure no errors were encountered during initialization. Instead of having to type the code in every time we want to create a graphics program, we can save this and open it when we need it:

|  |
| --- |
| #include<iostream.h>  #include<conio.h>  #include<graphics.h>  int main()  {  int gd,gm,ec;  gd=DETECT;  initgraph(&gd,&gm,” “);  ec=graphresult();  if(ec!=grOk)  {  cout<<”Error Initialization Graphics”<<”\n”<<grapherrormsg(ec)<<”\n”;  }  getch();  closegraph();  } |

There are certain BGI functions that we can use in our programs:

* getmaxx()
* getmaxy()

These functions return the maximum pixel co-ordinates in the horizontal a vertical direction, respectively. So (getmaxx(),getmaxy()) refers to the lower right corner of the graphics window.

* getmaxcolor()

This function returns the largest available color number.

For example: consider the following segment of code which displays 3500 “random” pixels in the graphics window:

|  |
| --- |
| #include<iostream.h>  #include<conio.h>  #include<graphics.h>  int main()  {  int xmax=getmaxx();  int ymax=getmaxy();  int maxcolor=getmaxcolor();  int color,pixel,x,y;  srand((unsigned)time(NULL));  for(pixel=0;pixel<3500;pixel++)  {  x=rand()%(xmax+1);  y=rand()%(ymax+1);  color=rand()%(maxcolor+1);  putpixel(x,y,color);  delay(15);  }  getch();  closegraph();  } |

* setviewport()

This function allows us to limit the drawing done to a specified rectangular portion of the graphics window.

For example: consider the following code:

|  |
| --- |
| setviewport(50,100,350,300,1);  setcolor(WHITE);  rectangle(0,0,299,199);  setcolor(RED);  line(25,50,425,50); |

This would draw a white rectangle around the viewport co-ordinates and a red line from (25,50) to (299,50) in viewport co-ordinates or from (75,150) to (350,150) in absolute co-ordinates. After a viewport is defined, all co-ordinates used will be relative to the viewport settings.

The following describes the function used in the above code:

setviewport(x1,y1,x2,y2,1/0)

(x1,y1) 🡪 Top Left co-ordinate of viewpoint

(x2,y2) 🡪 Bottom Right co-ordinate of viewpoint

1 🡪 Clipping is turned on

0 🡪 Clipping is turned off

* fillellipse(xCenter,yCenter,xRadius,yRadius);

The fillellipse function has four parameters:

(xCenter,yCenter) specifies the center point of the ellipse

xRadius specifies the horizontal radius

yRadius specifies the vertical radius

* fillpoly(numpoints,polypoints)

The fillpoly function and corresponding drawpoly function have two parameters:

polypoints should be an array containing the points of the polygon (must pass the first point in again as the last point in the polygon)

numpoints specifies how many actual

Example:

int polypoints[]={3,5,25,33,250,76,123,123,3,5};

drawpoly(5,polypoints);

**Drawing ellipses involves using the following function:**

ellipse(x,y,startAngle,endAngle,xRadius,yRadius);

(x,y) is the center point of the ellipse

Using startAngle and endAngle other than 0 and 360 allows us to draw a certain sector or portion of an ellipse

xRadius and yRadius are self explainatory

**Specifying Colors:**

If we want to change the color or pattern used for filling objects and regions, we can use the following functions:

* setfillstyle(int pattern,int color)

This function specifies the color to be used for the interior of objects.

There are 13 different patterns defined for the pattern of filling:

* EMPTY\_FILL(0)
* SOLID\_FILL(1)
* SLASH\_FILL(4)
* BK\_SLASH\_FILL(5)
* HATCH\_FILL(7)
* XHATCH\_FILL(8)
* position (x,y) sets the current position to (x,y)
* linerel (dX,dY) draws a line from the current position (CP) to (CP+dX,CP+dY)
* moverel (dX,dY) changes the current position (CP) to (CP+dX,CP+dY)

# setcolor() will specify the color used for the border of objects and normal drawing functions.

**Rectangles**

The rectangle function accepts four parameters: left, top, right, bottom. These parameters specify the upper left and the bottom right corners of the rectangle to be drawn.

Example:

rectangle(0,0,100,50);

**Circles**

The circle function accepts three parameters: xCenter, yCenter, radius. This is used to draw a circle with the center point being (xCenter,yCenter) and radius of radius.

Example:

circle(50,50,10);

The rectangle and circle function draws only the perimeter of the specified objects. This means that the objects are not filled with color when displayed. One way to fill an object or any closed region is to use the following function:

* floodfill(x,y,borderColor);

The floodfill function has three parameters: point (x,y) called the seed point, can be any point that is located inside the region to be filled; the borderColor value specifies the color of the border of the region to be filled, filling will be stop when borderColor is reached. The floodfill function can possibly “leak” or “spill” outside the region, perhaps filling the entire graphics window, if the (x,y) point will be selected is inside a non-closed region.

There are also two other fill functions, which draw and fill an object simultaneously:

* setcolor(color) 🡪 color is a constant or value describing the color of choice. Not sure on this but I believe there are 16 possible colors.
* rectangle(x1,y1,x2,y2) 🡪 (x1,y1) top left co-ordinate of rectangle; (x2,y2) bottom right co-ordinate of rectangle.
* line(x1,y1,x2,y2) 🡪 (x1,y1) starting co-ordinate of line; (x2,y2) ending co-ordinate of line.

In order to clear all or part of the graphics window, we can use following functions:

* cleardevice() 🡪 clears the entire graphics window.
* clearviewport() 🡪 clears the currently active viewport.
* Lines

The following functions may be used to draw lines:

line(x1,y1,x2,y2) 🡪 draws a line from (x1,y1) to (x2,y2)

moveto(x,y) 🡪 changes the current position to (x,y)

lineto(x,y) 🡪 draws a line in the current style and color from the current position to the specified.

The following section of code will draw a rectangle with a border of white and interior of red:

|  |
| --- |
| setcolor(WHITE);  rectangle(20,20,120,100);  setfillstyle(SOLID\_FILL, RED);  floodfill(21,21,WHITE); |