Main Page

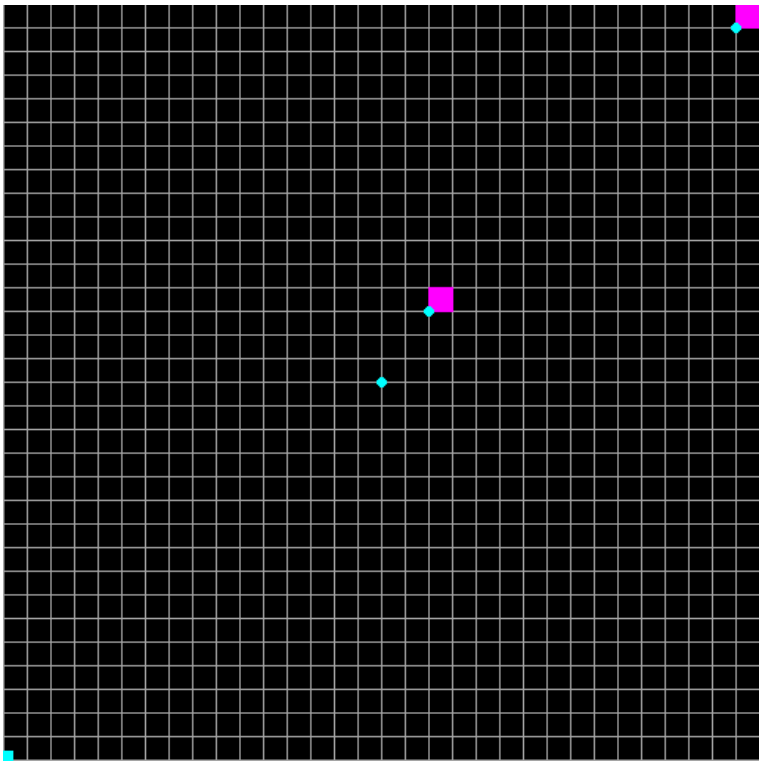
Lecture1/3: Scan Conversion, Triangle Filling, and Color Interpolation

- **Change checkerboard(Canvas& canvas) into a member function of Canvas**
- **Clear the Canvas to some clear color**

  Add the following member function to Canvas class

```
void Clear(RGB const& clear_col); //! Clear the whole canvas to the given clear color.
```

- **Specify a Pixel: (x,y) vs. (row, column)**



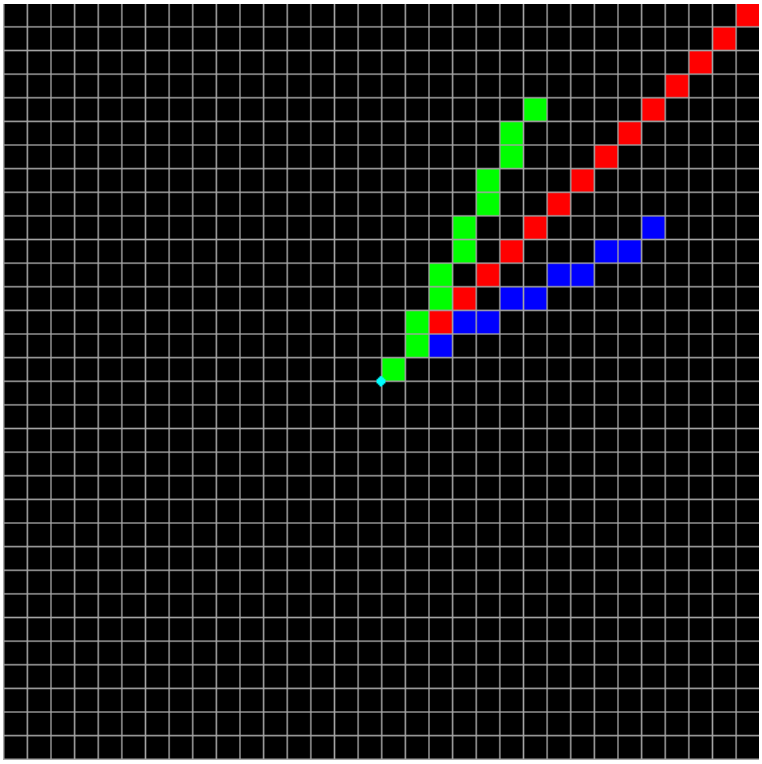
So far we use (row, column) to specify a pixel on the canvas. the other way is to use pair (x, y) to do the same thing.

Two important difference here.

  - (x=0, y=0) is the pixel in the center of the canvas
  - x, the first component, goes right, as column does, which is the second in pair (row, column)
  - y, the second component, goes up, as row does, which is the first in pair (row, column)
  - Now we assume vReso and hReso are both even. (you can force this in the constructor of Canvas).
  - the pixel on the lower left corner of the canvas is (x = -hReso/2, y = vReso/2)
  - the pixel on the upper right corner of the canvas is (x = hReso/2-1, y = vReso/2-1)

- **Scan Conversion Algorithm**

  Shown below are three lines, one in the first octant from point(0, 0) to point(11, 6), another in the second octant from point(0, 0) to point (6, 11), and the third one diagonal line. The first two lines are reflection to each other about the third line.



The steps in scan conversion algorithm

```
/*
1). swap the two end points to make sure the line goes to the right
2). translate the left point to the origin. Now the line is in either 1st or 4th quardant.
3). if the line goes down(in 4th quardant), flip the y value so that the line goes up. Now the line is in either 1st or 2nd octant.
4). if line has a slope great than 1 (2nd octant), flip it around the diagonal line.
5). Now the line is in the 1st octant; draw the line with simple line equation and round arithmetic. (NOT an incremental integer algorithm)
*/
```

- round()

  In the scan conversion algorithm below, you need round() which round a float number to its closest integer. Copy the following code into your xcpluscplus.h.

```
inline int round(float a)
{
  if(a>0)
    return int(a + .5);
  else
    return int(a - .5);
}
```

- **C++ Implementation**

  First copy this struct into Canvas.h right before the definition of Class Canvas.

```
struct ScannedResult
{
    ScannedResult(int X, int Y, RGB const& Col) : x(X), y(Y), col(Col) {  }

    bool operator<(ScannedResult const& rhs) const { return (y<rhs.y) || ( (y==rhs.y) && (x<rhs.x) ); }

    int x, y;
    RGB col;
};
```

  Now copy the scan conversion algorithm into your Canvas.cpp file.

```
/*
 Scan Conversion and Filling Triangle Algorithms
 */

// if swap_xy, flip the line around y=x line(i.e. the diagonal line).
// if flip_y,  flip the line around y=0 line(i.e, x axis)
static bool swap_xy = false, flip_y = false;


void Canvas :: scanLineSegment(int x1, int y1, RGB const col1,
                               int x2, int y2, RGB const col2,
                               set<ScannedResult>* output)
{
```

```
  assert( x2-x1 >= 0 && x2-x1 >= y2-y1 && y2-y1 >= 0);

  bool
    horizontal =  y1==y2,
    diagonal = y2-y1==x2-x1;

  float scale = 1.0 / (x2 - x1);

  RGB col = col1, col_step = (col2 - col1) * scale * .99999;

  int y = y1;
  float yy = y1, y_step = (y2 - y1) * scale;

  for(int x = x1; x <= x2; ++x, col += col_step)
  {
    int X(x), Y(y);
    if(swap_xy) swap(X,Y);
    if(flip_y)  Y = -Y;

    if(output)
      output->insert( ScannedResult(X, Y, col) );
    else
      Pixel(Y + vReso/2, X + hReso/2) = col;

    if( ! horizontal )
    {
      if( diagonal ) y++;
      else y = int(round( yy += y_step ));
    }
  }
}


void Canvas :: ScanLineSegment(int x1, int y1, RGB c1,
                               int x2, int y2, RGB c2,
                               set<ScannedResult>* output)
{
  if(x1 > x2)                                    // always scan convert from left to right.
  {
    swap(x1, x2); swap(y1, y2); swap(c1, c2);
  }
  if( (flip_y = y1 > y2) )                       // always scan convert from down to up.
  {
    y1 = -y1, y2 = -y2;
  }
  if( (swap_xy = y2-y1 > x2-x1) )                // and always scan convert a line with <= 45 deg to x-axis.
  {
    swap(x1, y1), swap(x2, y2);
  }
  scanLineSegment(x1, y1, c1,  x2, y2, c2, output);
}
```

- **Add WireTriangle() to Canvas**

```
    // assume the vertex 1,2,3 are in counter clockwise order
    void WireTriangle(int x1, int y1, RGB const col1,
                      int x2, int y2, RGB const col2,
                      int x3, int y3, RGB const col3);
```

and try to generate some triangles.

- **Add WireQuad() to Canvas.**

  Now do the same thing for a quad. Again in Quad(), you assume the vertex 1,2,3 and 4 are in counter clockwise order.

- **Filling Triangle Algorithm**

  This is done in two stages.

  First, scan convert all the three edges using ScanLineSegment(). However, the scanned result (the pixel position x and y, and the color at the pixel) is stored to some temporary data structure instead of writing to the canvas directly.

  Second, of all the scanned result above, any points of the same y value define a horizontal line inside the triangle. All such horizontal lines are drawn, and the triangle is filled.

  Now copy this into your Canvas.cpp file.

```
void Canvas :: FillTriangle(int x1, int y1, RGB const col1,
                            int x2, int y2, RGB const col2,
                            int x3, int y3, RGB const col3)

{
  set<ScannedResult> scanned_pnts;                // sorted in y val, and in x val if y val the same.
  ScanLineSegment(x1, y1, col1,  x2, y2, col2,  &scanned_pnts);
  ScanLineSegment(x2, y2, col2,  x3, y3, col3,  &scanned_pnts);
  ScanLineSegment(x3, y3, col3,  x1, y1, col1,  &scanned_pnts);
```

```
  int cur_yval = vReso / 2;                       // initialize to an invalid value.
  set<ScannedResult> same_yval;                   // of the scanned result.

  for (set<ScannedResult>::iterator it = scanned_pnts.begin(); it != scanned_pnts.end(); ++it)
  {
    int y = it->y;
    if(y != cur_yval)
    {
      if(same_yval.size())
      {
        set<ScannedResult>::iterator it1 = same_yval.begin(), it2 = --same_yval.end();
        ScanLineSegment(it1->x, cur_yval, it1->col,
                        it2->x, cur_yval, it2->col);

        same_yval.clear();
      }
      cur_yval = y;
    }
    same_yval.insert(*it);
  }
}
```

- **Now add SolidQuad() to Canvas**
- **1D Parametrization and Tesselation, WireCircle()**
  - Explicit equation of a circle

    A circle is the locus of point *(x,y)*, which has the same distance *r* to the central point *(cx, cy)*, or

    ```
    (x - cx)^2 + (y - cy)^2 = r^2
    ```

  - Parametric equations of a circle

    The points on the circle can also be specified by the angle *theta* it makes with the x-axis.

    ```
    x =  cx + r * cos(theta);
    y =  cy + r * sin(theta);
    ```

  1D tesselation is the method to approximate a smooth curve by polylines (consecutive small line segments). Now add WireCircle() to Canvas.

```
// draw total_segs line segments to approximate the given circle.
void WireCircle(int center_x, int center_y, float radius, int total_segs);
```

```
void Canvas :: WireCircle(int cx, int cy, float r, int segs)
{
  int x0 = round(cx + r) , y0 = cy;
  int x1, y1;
  float angle_step = 2 * pi / segs;
  float angle = angle_step;

  for(int i=0; i<segs; i++, angle += angle_step)
  {
    if(i==segs-1)
    {
      x1 = round(cx + r);
      y1 = cy;
    }
    else
    {
      x1 = round(cx + r * cos(angle));
      y1 = round(cy + r * sin(angle));
    }
    ScanLineSegment(x0, y0, Red, x1, y1, Red);

    x0 = x1, y0 = y1;
  }
}
```

- **Add SolidCircle() to Canvas**

  You should do two versions.

  - Based on triangulation approximation to the circle.

    ```
    void solidCircle(int center_x, int center_y, float radius, int total_segs);
    ```

  - Fill the circle directly

    ```
    void solidCircle(int center_x, int center_y, float radius);
    ```