Lignes, cercles et antialiasing
TD n°1

Ce premier TD a pour objectif de découvrir la base du programme qui vous est fournie et de la compléter avec quelques algorithmes vus en classe (segment de droite, cercle, antialiasing). Vous trouverez le projet à l'adresse suivante

http://hapco.limsi.fr/sites/hapco.limsi.fr/files/base-project.zip

Pour ce TD, il vous faudra exclusivement compléter les parties du code source où le commentaire suivant est marqué

```
// TODO => TP01 //
```

Le programme utilise deux bibliothèques : OpenGL et GLM. En ce qui concerne OpenGL, elle permet simplement d'accéder rapidement et facilement à des fonctions pour dessiner un tableau de pixels : vous n'aurez pas à modifier ou à utiliser cette bibliothèque.

GLM est une abbréviation de Open**GL** **M**athematics. Cette bibliothèque va nous fournir tous les outils mathématiques dont nous allons avoir besoin, à savoir des vecteurs, des matrices ainsi que tous les calculs qui s'y rapportent. Vous avez accès aux vecteurs à deux, trois ou quatre dimensions respectivement avec les classes `vec2`, `vec3` et `vec4` ainsi qu'aux matrices carrées de dimension quatre avec `mat4`. En ce qui concerne les vecteurs, vous pouvez accéder à chacune des composantes avec `v.x`, `v.y`, `v.z` et `v.w`[1]. Ensuite, vous avez accès aux opérateurs d'addition, de soustraction, de multiplication ou de division (seulement avec des scalaires). Enfin, le produit scalaire pour des vecteurs de même dimension est accessible avec la fonction `dot(v1, v2)`.

Exercice 1 *Implémentez l'algorithme de* BRESENHAM *pour le dessin d'un segment de droite. La fonction à compléter est la suivante*

```
void draw_line(vec2 p1, vec2 p2, vec3 c);
```

où ***p1*** *et* ***p2*** *sont les points aux extrémités du segment de droite et* `c` *est la couleur du segment de droite. N'oubliez pas de prévoir toutes les orientations possibles d'une droite.*

Pour dessiner un pixel, vous utiliserez la fonction suivante

```
void draw_pixel(vec2 p, vec3 c);
```

qui prend les coordonnées ***p*** *du point dessiné et la couleur* `c`*.*

Correction de l'exercice 1 – Pour dessiner une ligne, on peut utiliser l'algorithme simple (sans optimisation) vu en cours pour le faire évoluer vers un algorithme complet.

Source 1 – Fonction `draw_line` de base (sans optimisation)

```
void Window::draw_line(vec2 p1, vec2 p2, vec3 c)
{
  int x1 = (int)p1.x;
  int y1 = (int)p1.y;
  int x2 = (int)p2.x;
  int y2 = (int)p2.y;
  int dx = (int)abs(x2 - x1);
  int dy = (int)abs(y2 - y1);
  float a = (float)dy / (float)dx;
  float e = 0;
```

1. En considérant que le vecteur `v` a été déclaré avec `vec4 v`.

```
  int x = x1;
  int y = y1;
  while(x < x2)
  {
    draw_pixel(vec2(x,y), c);
    e += a;
    x++;
    if(e > 0.5)
    {
      y++;
      e--;
    }
  }
  draw_pixel(vec2(x,y), c);
}
```

Lorsqu'on cherche à étudier tous les cas, un premier problème est que cet algorithme ne produit pas le même résultat si on trace le segment de droite $[P_1, P_2]$ ou le segment de droite $[P_2, P_1]$. Afin de résoudre ce problème, on va toujours prendre le point avec la valeur minimale sur l'axe principal (x si $dx > dy$ ou y si $dx < dy$) ce qui revient à inverser les points. Attention cependant, cela entraîne plusieurs conséquences.

- Pour dessiner le segment de droite, on incrémentera toujours sur l'axe principal ;
- Sur l'axe secondaire, on incrémente si les p_1 et P_2 sont dans l'ordre sur cet axe ou on décrémente s'il sont dans le mauvais ordre ; cependant, si on doit inverser les points, l'incrémentation devient une décrémentation et vice-versa ;

De plus, la condition d'arrêt contenue dans le `while` n'est plus vraiment valable. Cependant, on connaît exactement le nombre d'itération nécessaire (avec dx) donc on peut remplacer la boucle `while` par une boucle `for`.

Puis, on va de nouveau optimiser le code avec les mêmes évolutions vues dans le cours pour aboutir à une version complète et optimisée de l'agorithme de Bresenham.

Source 2 – Fonction `draw_line` optimisée

```
void Window::draw_line(vec2 p1, vec2 p2, vec3 c)
{
  int x1 = (int)p1.x;
  int y1 = (int)p1.y;
  int x2 = (int)p2.x;
  int y2 = (int)p2.y;
  int dx = (int)abs(x2 - x1);
  int dy = (int)abs(y2 - y1);
  int xinc = (p2.x>p1.x)?1:-1;
  int yinc = (p2.y>p1.y)?1:-1;
  if(dx>dy)
  {
    int e = -dx;
    int x = (x1<x2)?x1:x2;
    int y = (x1<x2)?y1:y2;
    if(x2<x1)
    {
      yinc = -yinc;
    }
    for(int i=0; i<=dx; i++)
    {
      draw_pixel(vec2(x,y), c);
      e += 2*dy;
      x++;
```

```
      if(e > 0)
      {
        y += yinc;
        e -= 2*dx;
      }
    }
  }
  else
  {
    int e = -dy;
    int x = (y1<y2)?x1:x2;
    int y = (y1<y2)?y1:y2;
    if(y2<y1)
    {
      xinc = -xinc;
    }
    for(int i=0; i<=dy; i++)
    {
      draw_pixel(vec2(x,y), c);
      e += 2*dx;
      y++;
      if(e > 0)
      {
        x += xinc;
        e -= 2*dy;
      }
    }
  }
}
```

Exercice 2 *Ajoutez un antialiasing à votre moteur de rendu. Pour cela, vous allez vous baser sur la valeur de la variable* `sample` *qui est un attribut de la classe* `Window`*. Lorsque* `sample=1`*, le programme dessine le tableau* `pixels`*. Cependant, afin de créer un antialiasing, nous allons nous servir de* `pixels` *comme un tableau intermédiaire.*

Regardez dans la fonction `get_pixels()`*. Cette fonction est utilisée pour envoyer le tableau de pixels à afficher par OpenGL. Lorsque la valeur de* `sample` *est différente de 1, ce n'est plus* `pixels` *qui est donné à OpenGL mais* `pixels_final`*. La fonction* `antialiasing()` *aura donc pour fonction de remplir* `pixels_final` *en fonction de* `pixels` *qui aura été agrandi (d'un facteur* `sample` *en largeur et en hauteur).*

Étant donné que la fonction `draw_pixel()` *rempli le tableau* `pixels`*, vous pourrez utiliser l'autre fonction* `draw_pixel_sampled()` *pour remplir le second tableau* `pixels_final`*.*

Correction de l'exercice 2 –

Source 3 – Fonction antialiasing

```
void Window::antialiasing()
{
  vec3 color;
  for(int x=0; x<width-sample; x+=sample)
  {
    for(int y=0; y<height-sample; y+=sample)
    {
      for(int i=0; i<sample; i++)
```

```
      {
        for(int j=0; j<sample; j++)
        {
          color += get_pixel(vec2(x+i, y+j));
        }
      }
      color /= sample*sample;
      draw_pixel_sampled(vec2(x/sample, y/sample), color);
      color = vec3();
    }
  }
}
```

Source 4 – Fonction `draw_pixel_sampled`

```
void Window::draw_pixel_sampled(vec2 p, vec3 c)
{
  if(0<=p.x && p.x<width && 0<=p.y && p.y<height)
  {
    int indice = (p.x + p.y*width/sample) * 3;
    pixels_final[indice] = f2c(c.x);
    pixels_final[indice+1] = f2c(c.y);
    pixels_final[indice+2] = f2c(c.z);
  }
```

Exercice 3 *Implémentez l'algorithme de* Bresenham *pour le dessin d'un cercle avec la fonction suivante*

```
void draw_circle(vec2 center, unsigned int r, vec3 c);
```

où `center` *est le centre du cercle,* `r` *est le rayon et* `c` *est la couleur.*

On rappelle que l'algorithme ne dessine qu'un huitième du cercle. On utilisera la fonction suivante pour dessiner les huit parties du cercle en même temps

```
void draw_circle_parts(vec2 p, vec2 center, vec3 c);
```

où `p` *est le point actuellement dessiné,* `center` *est le centre du cercle et* `c` *est la couleur.*

Correction de l'exercice 3 –

Source 5 – Fonction `draw_circle`

```
void Window::draw_circle(vec2 center, unsigned int r, vec3 c)
{
  int x = 0;
  int y = r;
  int e = 1 - (double)r;
  draw_circle_parts(vec2(x, y), center, c);
  while(y>x)
  {
    if(e < 0)
```

```
    {
      e = e + 2*x + 3;
    }
    else
    {
      e = e + 2*x - 2*y + 5;
      y = y - 1;
    }
    draw_circle_parts(vec2(x, y), center, c);
    x = x + 1;
  }
}
```

Source 6 – Fonction `draw_circle_parts`

```
void Window::draw_circle_parts(vec2 p, vec2 center, vec3 c)
{
  draw_pixel(vec2(p.x, p.y)+center, c);
  draw_pixel(vec2(p.x, -p.y)+center, c);
  draw_pixel(vec2(-p.x, p.y)+center, c);
  draw_pixel(vec2(-p.x, -p.y)+center, c);
  draw_pixel(vec2(p.y, p.x)+center, c);
  draw_pixel(vec2(p.y, -p.x)+center, c);
  draw_pixel(vec2(-p.y, p.x)+center, c);
  draw_pixel(vec2(-p.y, -p.x)+center, c);
}
```