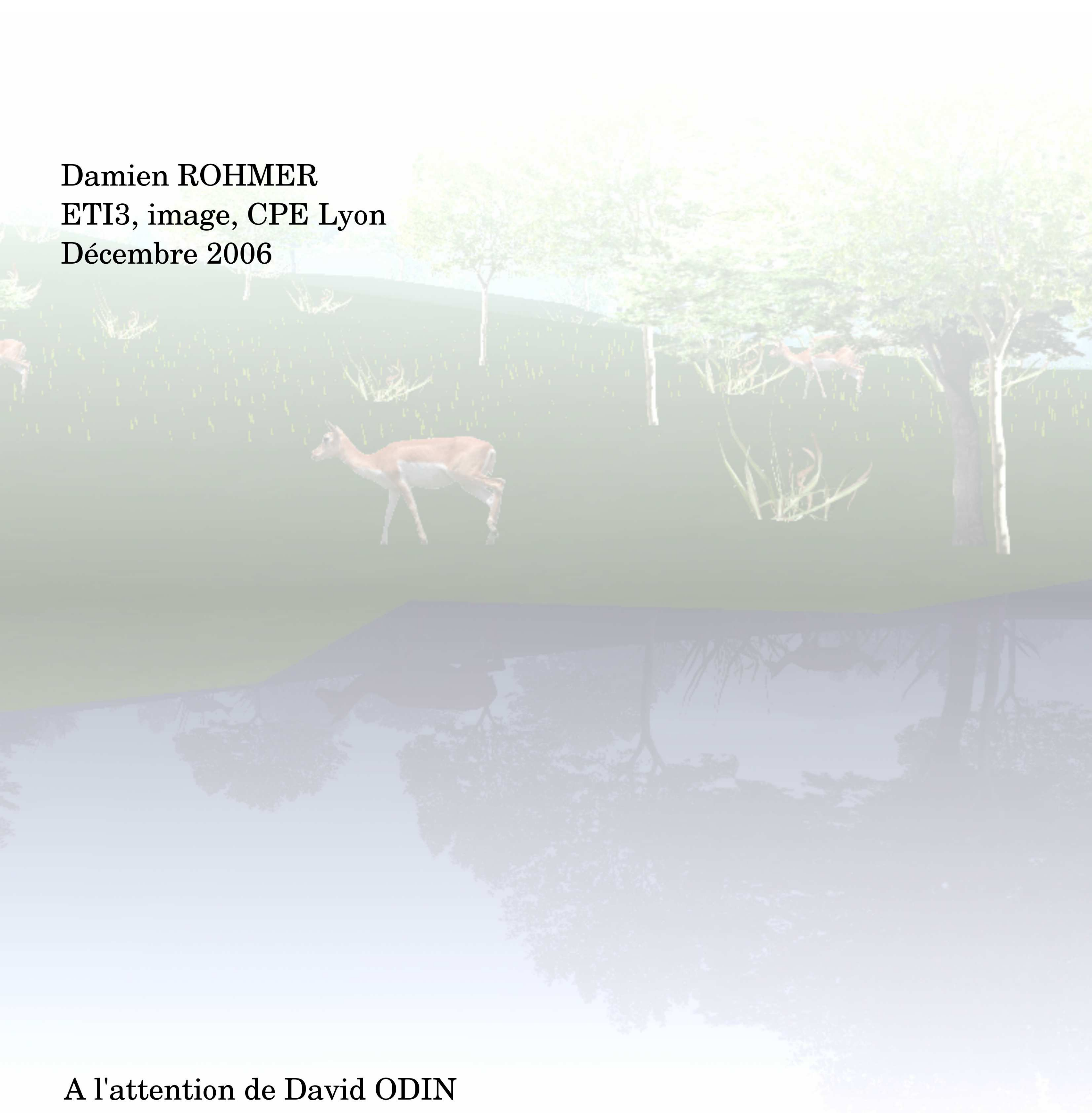
TP OpenGL

Rendu de paysage naturel avec
végétation et réflexion.



Damien ROHMER
ETI3, image, CPE Lyon
Décembre 2006

A l'attention de David ODIN

TP OpenGL

Damien Rohmer

Décembre 2006

Contents

1 Introduction **3**

2 Colline **4**
- 2.1 Creation du terrain . . . 4
  - 2.1.1 Théorie . . . 4
  - 2.1.2 Implémentation . . . 4
- 2.2 Coloration . . . 6
  - 2.2.1 Théorie . . . 6
  - 2.2.2 Implémentation . . . 7

3 Eau et reflexion **9**
- 3.1 Théorie . . . 9
- 3.2 Implémentation . . . 10

4 Sky box **13**
- 4.1 Théorie . . . 13
- 4.2 Implémentation . . . 15

5 Végétation **16**
- 5.1 Billboarding . . . 17
  - 5.1.1 Théorie . . . 17
  - 5.1.2 Implémentation . . . 18
- 5.2 Quads croisées . . . 20
  - 5.2.1 Théorie . . . 20
  - 5.2.2 Implémentation . . . 21
- 5.3 Herbe . . . 22
  - 5.3.1 Théorie . . . 22
  - 5.3.2 Implémentation . . . 23

6 Château **25**
- 6.1 Implémentation . . . 26

7 Objets 3D **27**
- 7.1 Théorie . . . 27
- 7.2 Implémentation . . . 28

8 Épée et déplacement **30**
- 8.1 Théorie . . . 30
- 8.2 Implémentation . . . 31

9 Map **32**
- 9.1 Théorie . . . 32
- 9.2 Implémentation . . . 33

10 Brouillard **34**
- 10.1 Théorie . . . 34
- 10.2 Implémentation . . . 34

11 Discussion **35**
11.1 Scène 35
11.2 Problèmes 37
11.2.1 Billboards 37
11.2.2 Quads croisées 37
11.2.3 Sky Box 38
11.2.4 Brouillard 39
11.2.5 Textures 40
11.2.6 Divers points 40

12 Conclusion **41**

1 Introduction

Ce TP concerne l'utilisation de techniques simples d'affichage permettant de simuler l'apparence de scènes naturelles. Le but principal est d'implémenter dans la scène une réflexion et de la végétation. Pour cela, on choisi de créer une scène naturelle avec des collines entourée d'eau. Les collines sont recouvertes par différent types de végétations.

On s'interessera particulièrement aux différentes techniques utilisées. Les bases de la mise en place de textures, chargement de l'image, ou affichage de surface 3D ne seront pas détaillés dans ce rapport.

Dans ce TP, le programme n'est pas spécialement optimisé au sense de l'espace mémoire utilisé. En effet, les textures prise en compte seront souvent de taille trop importante par rapport à leur utilisation. Cepandant cela n'est pas l'objet principale de ce programme.

La scène utilisées possèdera certains détails ayant chacun un aspect plus ou moins réaliste. La plupart d'entre eux seront surtout une excuse pour utiliser certaines techniques d'OpenGL. On pourra noter parmis les principales:

- Reflexion par rotation de la scène.
- Effet de transparence par blending et gestion de l'ordre d'affichage.
- Mise en place de billboards.
- Utilisation de quads/triangles texturés (croisés ou non) simulant ici de la végétation.
- Utilisation du multitexturing.
- Création d'animation de feu et d'épée par sprites.
- Utilisation d'une sky box.
- Addition d'objets non paramétrique 3D plus complexe.
- Déplacement sur un terrain non plat.
- Utilisation du viewport afin d'afficher plusieurs scène en même temps.

Nous verrons étapes par étapes les techniques utilisées pour la création de la scène. Ainsi, nous expliquerons la création de la colline de base sur lequel le personnage pourra se déplacer. L'eau et la technique de reflexion seront ensuite introduite. La sky box donnant l'impression d'horizon à l'infini sera mise en place, puis nous expliqueront plus en détail les différentes méthodes utilisées pour la végétation mise en place dans la scène. Enfin nous verrons rapidement quelques détails du paysage avec le chateau, les objets 3D, le sprite de l'épée, l'affichage de la carte en surimpression puis finalement, la mise en place du brouillard. Nous feront, en dernier point, un état des lieux sur certains avantages et inconvénients de l'utilisation de ces techniques.

2 Colline

2.1 Creation du terrain

2.1.1 Théorie

La colline est tout d'abord implémenté sous forme de height field (ou height map). Pour cela, on considère une fonction $f : (x, z) \mapsto f(x, z)$, et l'on associe une altitude y au point (x, z) donné par f. D'une facon numérique, la fonction f nous est fournie par une image à niveaux de gris. Chaque point (x, z) nous est donné sur une grille $2D$ de $N_x \times N_z$ régulièrement espacé par pas de $\Delta_x \times \Delta_z$. À chaque position de la grille, on y associe alors l'altitude correspondante des points de l'image.

Dans le cas présent, l'image montrée en fig. 1 est utilisée. Cette image est de taille 512×512,

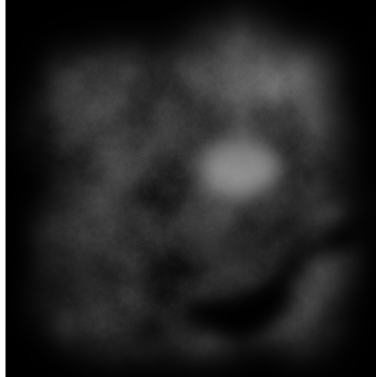


Fig. 1: Image des hauteurs utilisée pour la mise en place du terrain.

cepandant, ce niveau de résolution n'est pas nécessaire. Dans le cas présent, le terrain est déja grand et n'utilise qu'une résolution de 50×50 points.

On peux visualiser en fig. 2 le terrain en fil de fer sur lequel la scène va se dérouler. L'image montre

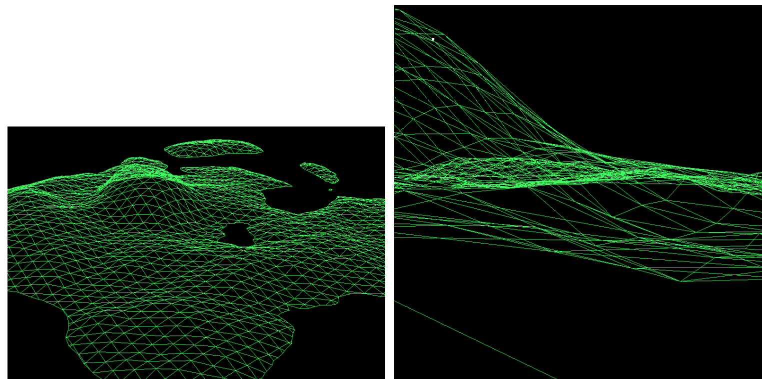


Fig. 2: Image du terrain vue en fil de fer. L'image de gauche représente une vue de haut. L"image de droite représente le terrain au niveau du sol lorsque l'on se déplace dessus.

que l'on coupe le terrain pour des altitudes inférieures au niveau de la mer. Cette coupure peux se réaliser directement par un plan au niveau OpenGL. En considérant la mer située à la hauteur h, le plan de coupure est défini par l'équation: $y - h = 0$.

Enfin, afin de pouvoir se déplacer sur le terrain, il faut connaître la hauteur de celui ci en n'importe quel point (x, z). Pour cela, on implémente une interpolation bilinéaire sur la position afin de contraindre le déplacement à rester sur la surface.

2.1.2 Implémentation

Le chargement des textures se réalise grace à la classe "Texture". Comme indiqué dans l'introduction, on ne détaille pas l'implémentation de cette classe. Le terrain en lui même est implémenté par la classe

"Ground". On charge une image de type "png"[1]. Comme on l'a précisé initiallement, le code n'etant pas optimisé pour l'utilisation mémoire, on charge une image couleur et non uniquement noire et blanc pour la height map. Afin de considerer une hauteur scalaire, on prend uniquement en compte la composante rouge.

La classe Ground est implémenté sous cette forme:

```
//Height map
class Ground
{
 public:

  Ground();
  ~Ground();


  int set_height_field_texture(const char* texture_path);
  int build(int N_x,int N_y,const char* height_field_path);
  int draw();

  float get_height(float _x,float _y);
  int get_normal(float _x,float _y,float *n);
  int set_scale(float scale_x,float scale_y,float scale_z);

  float get_size_x();
  float get_size_y();

 private:

  int destroy();
  int create_call_list();

  //heigth map
  int N_x,N_y;//size
  Mesh_tool mesh;//full mesh
  float *height_map;//height only
  float scale[3];

  //texture
  GLuint texture_number[2];
  Texture texture;

  GLuint list_number;
};
```

Les hauteurs sont stockées en float dans la variable "height map". Le terrain totale (x,y et z) est stocké sous forme de triangles par la variable "mesh" de la classe "Mesh tool". Cette classe permet de traiter des triangles en les sauvegardant sous la forme: vertex, normal et connectivitée. Cette classe ne sera pas non plus développée dans ce rapport car elle ne réalise que le contrôle "bas niveau" d'une liste de triangle et la création des normales. La fonction de construction des triangles est appelée par la méthode "build" de la classe "Ground". La méthode de construction des triangle est la suivante:

```
  for(k_x=0;k_x<N_x;k_x++)
    for(k_y=0;k_y<N_y;k_y++)
      {
       //interpolate the value of the texture
       height_field.get_linear_texel((float)k_x/N_x,(float)k_y/N_y,interpolated_texel);
       //take only red component in account for height field
       current_vertex[0] = (float)k_x/(N_x-1)*scale[0];
```

[1]La librairie png est donc necessaire

```
      current_vertex[1] = (float)interpolated_texel[0]/256*scale[1];
      current_vertex[2] = (float)k_y/(N_y-1)*scale[2];

      //fill the mesh with the vertex
      mesh.fill_vertex(k_x,k_y,N_x,current_vertex);
      //fill also the height map
      height_map[k_x+k_y*N_x] = current_vertex[1];
     }
```

Cette fonction de construction n'est appelée qu'une seule fois au début du programme.

Finalement, on peux implémenter la fonction de coupure de plan simplement par:

```
  //cut surface
  //plane y-WATER_LEVEL=0
  clip[0] = 0.0;
  clip[1] = 1;
  clip[2] = 0;
  clip[3] = -WATER_LEVEL;

  glEnable (GL_CLIP_PLANE0);
  glClipPlane (GL_CLIP_PLANE0, clip);
```

Et l'interpolation bilinéaire afin de contraindre le déplacement se réalise par la méthode "get height":

```
float Ground::get_height(float _x,float _y)
{
  float u_0=0.0,u_1=0.0;
  int k_0=0,k_1=0;

  u_0 = _x*(N_x-1)/scale[0];
  u_1 = _y*(N_y-1)/scale[2];

  k_0 = (int)u_0;
  k_1 = (int)u_1;

  float t_x=(u_0-k_0);
  float t_y=(u_1-k_1);

  int X00 = (k_0+0) + (k_1+0)*N_x;
  int X01 = (k_0+1) + (k_1+0)*N_x;
  int X10 = (k_0+0) + (k_1+1)*N_x;
  int X11 = (k_0+1) + (k_1+1)*N_x;

  float h=0.0;
  h =   t_x *     t_y *height_map[X11]+
    (1-t_x) *     t_y *height_map[X10]+
       t_x  * (1-t_y) *height_map[X01]+
    (1-t_x) * (1-t_y) *height_map[X00];

  if(h>WATER_LEVEL)
    return h;
  else
    return WATER_LEVEL;
}
```

2.2 Coloration

2.2.1 Théorie

On colore ensuite le terrain par plaquage de textures. Pour cela, on va utiliser deux textures d'echelles différentes.

- Une première texture à large échelle dont la couleur est liée à la hauteur du terrain: vert sur les faibles hauteurs et marron/gris sur les hauteurs plus importante.
- Une seconde texture à faible échelle répétée un grand nombre de fois donnant l'impression de détails sur le terrain.

Les deux textures sont affichées en fig. 3. Celle ci sont plaquées sur le terrain par multitexturing.

Fig. 3: Textures pour le terrain. La texture de gauche représente la texture à large échelle alors que celle de droite représente celle des détails à faible échelle.

L'ensemble de l'affichage est stocké dans une liste d'affichage car le terrain est fixe pendant tout le parcours de la scène.

2.2.2 Implémentation

On utilise la technique du multitexturing directement. On pourrait utiliser la méthode du blending par transparence, cepandant cela necessiterait l'envoie de la géométrie du terrain 2 fois dans le pipe-line de rendu. Le multitexturing permet de mixer les couleurs tout en n'envoyant qu'une seule fois chaque vertex. La mise en place de ces deux textures par vertex est réalisée dans la classe "Ground" lors de la création de la liste d'affichage:

```
//[texture_1 texture_2]
float *multitexturing=mesh.get_texture_coordinate();

glEnableClientState(GL_VERTEX_ARRAY);
glEnableClientState(GL_NORMAL_ARRAY);

//enable multitexturing
glActiveTextureARB(GL_TEXTURE0_ARB);
glBindTexture(GL_TEXTURE_2D,texture_number[0]);
glEnable(GL_TEXTURE_2D);
glClientActiveTextureARB(GL_TEXTURE0_ARB);
glTexCoordPointer(2,GL_FLOAT,0,&multitexturing[0]);
glEnableClientState(GL_TEXTURE_COORD_ARRAY);

glActiveTextureARB(GL_TEXTURE1_ARB);
glBindTexture(GL_TEXTURE_2D,texture_number[1]);
glEnable(GL_TEXTURE_2D);
glClientActiveTextureARB(GL_TEXTURE1_ARB);
glTexCoordPointer(2,GL_FLOAT,0,&multitexturing[2*N_x*N_y]);
glEnableClientState(GL_TEXTURE_COORD_ARRAY);

//set vertex and normal
glVertexPointer(3,GL_FLOAT,0,mesh.get_vertex());
glNormalPointer(GL_FLOAT,0,mesh.get_normal());
```

```
//draw
glDrawElements(GL_TRIANGLES,mesh.get_N_triangle()*3,GL_UNSIGNED_INT,mesh.get_connectivity());

//disable
glDisableClientState(GL_VERTEX_ARRAY);
glDisableClientState(GL_NORMAL_ARRAY);

glActiveTextureARB(GL_TEXTURE1_ARB);
glClientActiveTextureARB(GL_TEXTURE1_ARB);
glDisableClientState(GL_TEXTURE_COORD_ARRAY);
glDisable(GL_TEXTURE_2D);

glActiveTextureARB(GL_TEXTURE0_ARB);
glClientActiveTextureARB(GL_TEXTURE0_ARB);
glDisableClientState(GL_TEXTURE_COORD_ARRAY);
glDisable(GL_TEXTURE_2D);
```

On peux alors observer pour chaque application de texture le résultat en fig. 4 Une fois la liste d'execution

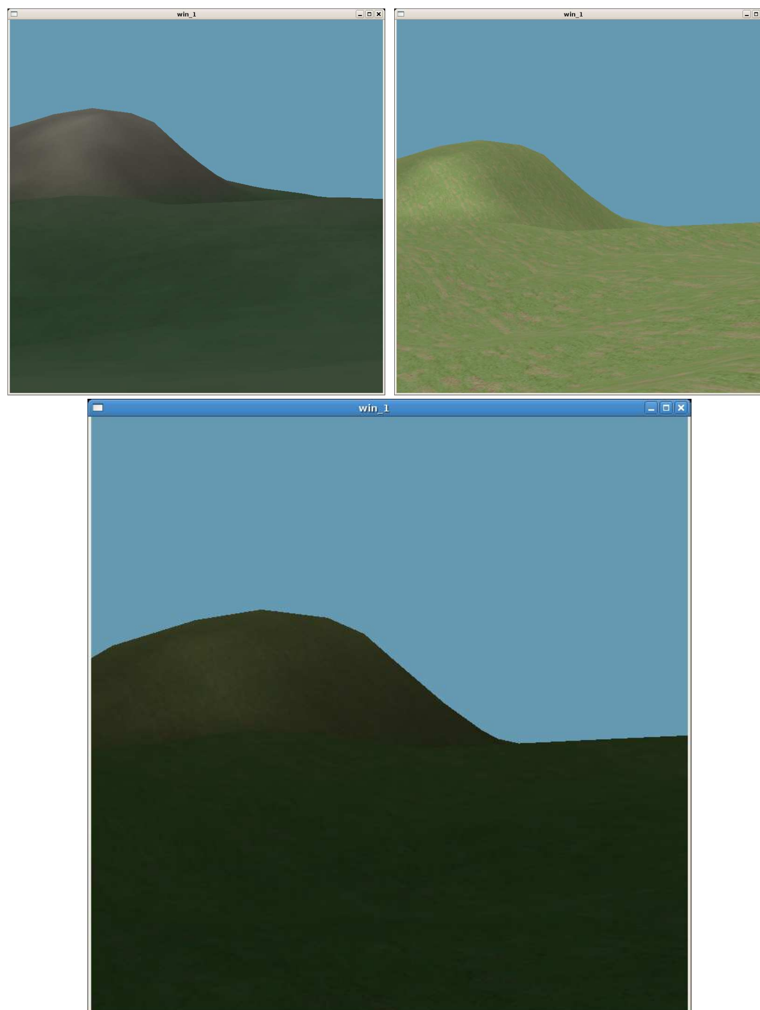


Fig. 4: La figure de gauche représente l'application de la texture large échelle mettant en valeure les montagnes et vallée par le changement de couleur. la seconde texture représente la texture à faible échelle donnant des détails se répétant et n'etant pas lié au terrain. Enfin la troisième texture représente le mélange entre les deux.

créée par l'appel suivant au début du programme:

```
//create the drawing list
if(glIsList(list_number))
```

```
  glDeleteLists(list_number,1);
 list_number = glGenLists(1);
 glNewList(list_number,GL_COMPILE);

 ... [code d'appel des vertex]

 glEndList();
```

Il suffit d'appeler à chaque affichage du programme principale cette liste d'affichage. La méthode "draw" se résume donc au stricte minimum afin d'obtenir le terrain complet de la fig. 5:

```
int Ground::draw()
{
  glCallList(list_number);
  return 0;
}
```

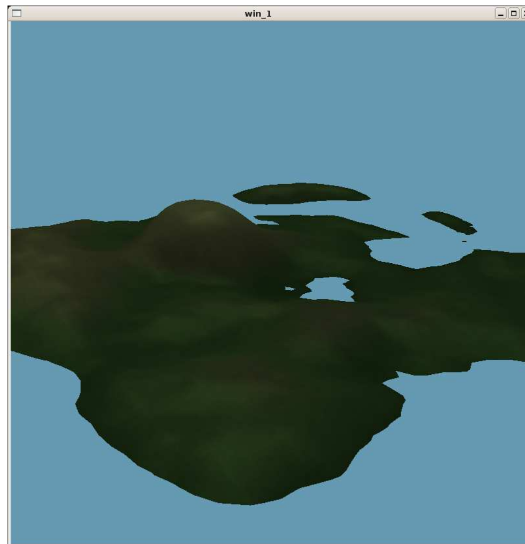


Fig. 5: Terrain complet texturé.

3 Eau et reflexion

3.1 Théorie

On applique alors la reflexion afin de donner une impression d'eau. Pour cela, on va afficher le terrain à l'envers. Il suffit pour cela de définir une matrice de reflection à multiplier sur la matrice du monde avant l'affichage de la reflection. La matrice de symétrie par rapport à y est donc définie par

$$M = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Le terrain est ainsi affiché deux fois (un unique envois dans le pipe line pourrait désormais se réaliser à l'aide d'un shader en affichant pour chaque vertex envoyé, 2 vertex sortant (création de vertex dans un shader désormais possible)), l'un étant à sa position et l'autre étant le symétrique). L'affichage du terrain et de son symétrique est montré en fig. 6

Une fois la reflection réalisé, on peux fournir un aspect de transparence par blending. Il suffit pour cela de placer, à la hauteur de l'eau h, un plan de taille importante, bleuté et légèrement transparent

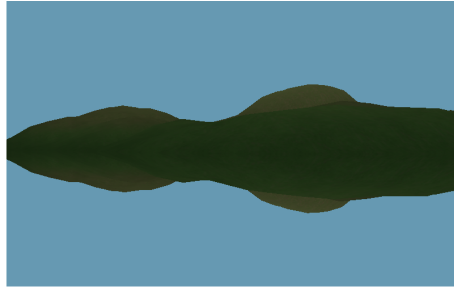


Fig. 6: Terrain et son image reflchie.

afin de donner l'impression d'eau. Comme le plan est transparent, il est necessaire de l'afficher après le terrain réfléchi.

Enfin, l'eau apparaît alors comme plat et statique. Afin de donner l'imporession de mouvement différentes solutions sont possibles. On pourrait modifier la positions des vertex du plan afin de simuler des vaguelettes. Cette solution est cepandant lourde d'un point de vue rapidité car elle necessite l'envoi d'un grand nombre de vertex. On peux considerer une solution plus simple ne necessitant que très peux de vertex. Un plan est défini par les 4 sommets d'un carré. Sur celui ci, on plaque une texture d'eau montré en fig. 7. Cette texture[2] répétée sur le grand plan donnera l'impression de non uniformité de la surface.

Fig. 7: Texture d'eau utilisée.

Ce plan est cepandant toujours statique. Afin de donner une impression de mouvement, il est alors necessaire d'augmenter le nombre de vertex sur le plan. (Cette étape pourrait être évitée par l'utilisation d'un fragment shader). On va alors donner l'impression de mouvement en modifiant les coordonnées de textures appliqué à chaque vertex. En appliquant des fonctions périodiques et continues sur les coordonnées de textures, on peux alors donner une impression de déformation de la surface alors que l'ensemble des vertex sont fixes.

Enfin, on peux donner une impression accentuée de profondeur en utilisant la technique de bump mapping. Pour cela, on déforme les normales en chaque vertex. En appliquant alors de la lumière sur la surface sensée être plane, la déformation des normales va donner une impression de relief. Il est facile également de modifier les normales de facon dynamique en rendant dépendant du temps ces modifications.

3.2 Implémentation

La surface d'eau est implémenté par la classe "Water surface"

```
class Water_surface
{
 public:
```

[2] texture réalisé "à la main" avec Gimp

```
  Water_surface();
  ~Water_surface();

  int set_size(int _N_x,int _N_y,float scale_1,float scale_2);
  int set_texture(const char* file_name);
  int draw();
  int animate(int kt);

 private:

  int destroy();

  int N_x,N_y;

  GLuint texture_number;
  float *vertex;
  float *normal;
  float *texture_coord;
  int *connectivity;
};
```

La creation des vertex se réalise de la facon suivante:

```
      //fill the pointers
      int k_x=0,k_y=0;
      for(k_x=0;k_x<N_x;k_x++)
       {
          for(k_y=0;k_y<N_y;k_y++)
            {
              vertex[3*(k_x+k_y*N_x)+0] = (float)k_x/(N_x-1)*scale_1;
              vertex[3*(k_x+k_y*N_x)+1] = WATER_LEVEL;
              vertex[3*(k_x+k_y*N_x)+2] = (float)k_y/(N_y-1)*scale_2;

              normal[3*(k_x+k_y*N_x)+0] = 0;
              normal[3*(k_x+k_y*N_x)+1] = 1;
              normal[3*(k_x+k_y*N_x)+2] = 0;

              texture_coord[2*(k_x+k_y*N_x)+0] = (float)k_x/(N_x-1)*LAMBDA_WATER;
              texture_coord[2*(k_x+k_y*N_x)+1] = (float)k_y/(N_y-1)*LAMBDA_WATER;
            }
       }
```

On active également le blending afin de permettre la transparence. La mise en place de la transparence est réalisé de la facon suivante:

```
  //set color
  glColor4f(0.2,0.2,0.4,0.7);
  //enable texture
  glEnable(GL_BLEND);
  glBlendFunc (GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);


...[affichage]

  glDisable(GL_BLEND);
```

L'affichage de la surface bleue est montré en fig. 8.

Ensuite, la texture est appliquée simplement afin d'obtenir le résultat de la fig. 9 Finalement, on anime l'eau en modifiant la texture. Pour cela, on fait appel à chaque affichage à la méthode "animate" prennant en paramètre le temps actuel. L'animation se réalise avec les fonctions suivantes:

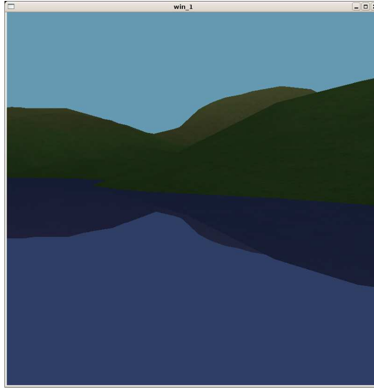


Fig. 8: Image de l'application de la surface blue d'eau.

Fig. 9: Surface d'eau avec l'application de la texture.

```
for(k_x=0;k_x<N_x;k_x++)
  {
    px=(float)k_x/(N_x-1);
    for(k_y=0;k_y<N_y;k_y++)
      {
        py=(float)k_y/(N_y-1);

        //change normal
        normal[3*(k_x+k_y*N_x)+0]=0.25*powf(sin(2*t-py*2*PI),2);
        normal[3*(k_x+k_y*N_x)+2]=0.25*2*powf(sin(3*t-px*2*PI),2);

        //renormalization
        for(norm=0,k_dim=0;k_dim<3;k_dim++)
          norm += normal[3*(k_x+k_y*N_x)+k_dim]*normal[3*(k_x+k_y*N_x)+k_dim];
        norm = powf(norm,(float)0.5);
        for(k_dim=0;k_dim<3;normal[3*(k_x+k_y*N_x)+k_dim]/=norm,k_dim++);

        //change texture
        texture_coord[2*(k_x+k_y*N_x)+0] = LAMBDA_WATER*(
          px+0.1/10*cos(10*t-px*(2*PI)*3)+0.1*cos(2*t-py*PI*2));
        texture_coord[2*(k_x+k_y*N_x)+1] = LAMBDA_WATER*(
          py+0.1/10*cos(10*t-py*(2*PI)*3)+0.05*cos(8*t-py*2*PI));

      }
  }
```

Afin d'évoluer le temps au cours de l'affichage, on incrémente une variable globale *kt* dans le programme principal lorsque l'écran se raffraichit:

```
static void idle_callback()
{
  kt++;
  glutPostRedisplay();
}
```

On créé des "patchs" rectangulaires d'eau de taille 10×10 et possédant 40×40 points. La résolution est donc faible par rapport au terrain, l'affichage necessite ainsi peu de vertex. Afin de couvrir l'ensemble du terrain, et même au dela pour obtenir un effet d'horizon, la surface d'eau doit être encore plus étendue. Pour cela, on accole plusieurs "patchs" les uns à coté des autres (il faut prendre soins lors des déformations de textures d'obtenir des périodicitées correctes afin de pouvoir accoler les quads les uns les autres). La fig. 10 montre les différentes surfaces en fil de fer afin de visualiser l'espacement des différents vertex.

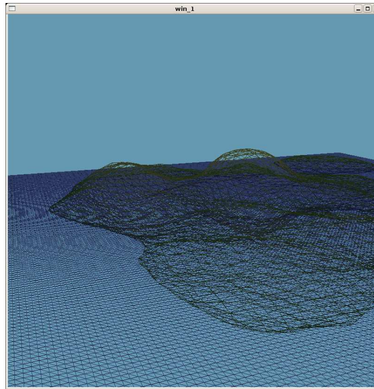


Fig. 10: Image de du terrain et de la surface d'eau en fil de fer.

Lors de l'appel à la méthode d'affichage, on utilise la fonction suivante:

```
//draw many patch
int k_x=0,k_y=0;
for(k_x=-1;k_x<=3;k_x++)
  for(k_y=-1;k_y<=3;k_y++)
    {
      glPushMatrix();
      glTranslatef(10*k_x,0,10*k_y);
      glDrawElements(GL_TRIANGLES,3*2*(N_x-1)*(N_y-1),GL_UNSIGNED_INT,connectivity);
      glPopMatrix();
    }
```

Enfin l'affichage de la surface d'eau se réalise par les appels suivants dans la fonction principale:

```
water.animate(kt);
water.draw();
```

Les résultats obtenues sont affichées en fig. 11 Comme l'eau est animée, elle doit être recalculée à chaque animation. On ne peux donc pas creer une fois pour toute une liste d'affichage, mais on est obligé d'envoyer à chaque affichage l'ensemble des vertex à la carte graphique, ce qui rend l'affichage plus lent (problème pouvant réglé par un vertex shader).

4 Sky box

4.1 Théorie

On met en place une technique appelée sky box afin d'avoir l'impression d'horizon à l'infini. Pour cela, on considère une boîte rectangulaire entourant la position courante. La boîte est située à une distance

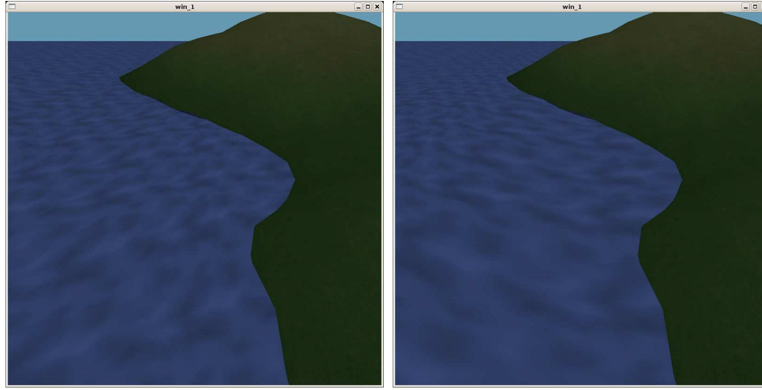


Fig. 11: Image de l'animation de la surface d'eau à 2 périodes différentes.

constante du personnage ce qui permet d'obtenir l'impression d'horizon fixe lors du déplacement. Afin d'avoir la vision d'un paysage, on plaque sur les faces une texture organisée comme le patron d'une boîte comme le montre la fig. 12. Cette texture organisée en 5 morceaux représente la partie supérieure du

Fig. 12: Image de la texture de la sky box.

parallélépipède. La partie inférieure n'étant pas importante car celle ci est recouverte d'eau.

Cette facon d'obtenir l'impression d'horizon avec des images de fond réaliste coûte très peu au niveau du temps de calcul car il ne necessite l'envoi que d'un cube et de sa texture associée (les fragments associées dans la carte graphique sont par contre nombreux car le fond peux prendre une place importante de la scène). Une vue éloignée de la sky box est montré en fig. 13 (ce qui n'arrive jamais en théorie, car celle ci est centrée sur le personnage). La sky box choisie est fixe (on pourrait penser à l'animer dans le cas

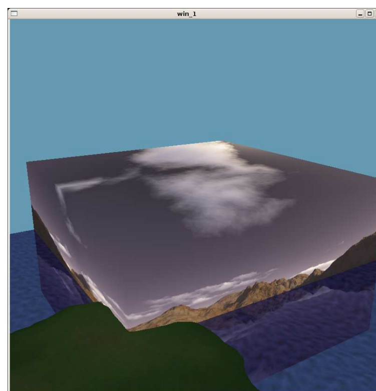


Fig. 13: Image de la sky box vue de loin.

de nuages en modifiant les coordonnées de textures). On peux donc obtenir un affichage très efficace en utilisant l'appel à une liste d'affichage.

4.2 Implémentation

La gestion de cette sky box est réalisé par la classe "Sky box".

```
class Sky_box
{
 public:
  Sky_box();
  ~Sky_box();

  int set_texture(const char* file_name);
  int draw();

 private:

  int build_draw_list();

  //size of the sky box
  int D,H;
  //textures
  GLuint texture_number[5];
  //draw list
  GLuint draw_list;
};
```

La création des cinq murs se réalise très facilement à l'aide de quads.

```
  glEnable(GL_TEXTURE_2D);

  //color
  glColor4f(1.0,1.0,1.0,1.0);

  //face 1
  glBindTexture (GL_TEXTURE_2D, texture_number[0]);
  glBegin(GL_QUADS);
  glTexCoord2f(0.0,1.0); glVertex3f(-D, 0,-D);
  glTexCoord2f(1.0,1.0); glVertex3f(-D, 0, D);
  glTexCoord2f(1.0,0.0); glVertex3f(-D, H, D);
  glTexCoord2f(0.0,0.0); glVertex3f(-D, H,-D);
  glEnd();

  //face 2
  glBindTexture (GL_TEXTURE_2D, texture_number[1]);
  glBegin(GL_QUADS);
  glTexCoord2f(1.0,1.0); glVertex3f(-D, 0,-D);
  glTexCoord2f(0.0,1.0); glVertex3f( D, 0,-D);
  glTexCoord2f(0.0,0.0); glVertex3f( D, H,-D);
  glTexCoord2f(1.0,0.0); glVertex3f(-D, H,-D);
  glEnd();

  //face 3
  glBindTexture (GL_TEXTURE_2D, texture_number[2]);
  glBegin(GL_QUADS);
  glTexCoord2f(1.0,1.0); glVertex3f( D, 0,-D);
  glTexCoord2f(0.0,1.0); glVertex3f( D, 0, D);
  glTexCoord2f(0.0,0.0); glVertex3f( D, H, D);
  glTexCoord2f(1.0,0.0); glVertex3f( D, H,-D);
  glEnd();

  //face 4
```

```
glBindTexture (GL_TEXTURE_2D, texture_number[3]);
glBegin(GL_QUADS);
glTexCoord2f(0.0,1.0); glVertex3f(-D, 0, D);
glTexCoord2f(1.0,1.0); glVertex3f( D, 0, D);
glTexCoord2f(1.0,0.0); glVertex3f( D, H, D);
glTexCoord2f(0.0,0.0); glVertex3f(-D, H, D);
glEnd();

//face 5: top
glBindTexture (GL_TEXTURE_2D, texture_number[4]);
glBegin(GL_QUADS);
glTexCoord2f(0.0,1.0); glVertex3f(-D, H,-D);
glTexCoord2f(0.0,0.0); glVertex3f( D, H,-D);
glTexCoord2f(1.0,0.0); glVertex3f( D, H, D);
glTexCoord2f(1.0,1.0); glVertex3f(-D, H, D);
glEnd();

glDisable(GL_TEXTURE_2D);
```

Enfin, pour rester constamment autour du personnage lors de l'affichage, on effectue juste une translation du cube avant l'affichage dans le programme principale.

```
//sky_box
glPushMatrix();
glTranslatef(position.get_x(),WATER_LEVEL,position.get_z());
sky_box.draw();
glPopMatrix();
```

Afin d'obtenir le reflet des nuages dans l'eau ainsi que de l'horizon, on dessine également cette sky box deux fois. Le résultat est affiché en fig. 14.

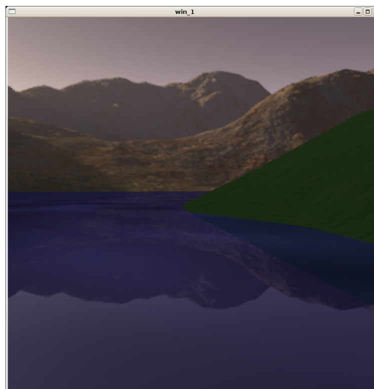


Fig. 14: Image de la scène avec la sky box réfléchie dans l'eau au loin.

5 Végétation

La végétation de la scène est composée de différentes parties afin de mettre en oeuvre différentes techniques ayant chacune avantages et inconvénients. Les différentes méthodes utilisées seront:

- Les billboards pour les biches et les arbres.
- Des quads fixes croisées pour les grandes touffes d'herbes.
- De simples triangles pour les petites herbes.

La végétation pose des problèmes spécifiques pour son rendu. En effet, pour obtenir le rendu d'une scène naturelle réaliste, chaque élément de végétation possède de nombreux détails; et de plus ces éléments sont très nombreux. La visualisation en temps réel de ce type de scène est donc complexe.

Pour cela, on utilise des techniques donnant l'illusion de détails et de necessitant que peux de calculs pour la machine.

5.1 Billboarding

5.1.1 Théorie

La première méthode utilisée est celle du billboarding. Elle consiste à afficher un rectangle faisant constamment face à la caméra. On plaque alors une texture représentant la végétation sur ces rectangles. Les textures peuvent représenter des végétaux complexes tels que des arbres entiers. Par contre, il est necessaire d'utiliser la transparence autour de la partie de la végétation sur la texture.

On utilisera trois types de textures[3] différentes sur les billboards. Pour chaque texture, le blanc est remplacé par de la transparence (canal alpha à 1). Les trois textures sont montrés en fig. 15.

Fig. 15: Texture de biches et d'arbres utilisées sur les billboards.

On va donc pouvoir positionner ces rectangles faisant face à la caméra en différents endroits de la surface du sol. Le problème majeur étant l'utilisation de la transparence. En effet, les éléments sont positionnées à des endroits aléatoires, et l'utilisation de la transparence nous force à désactiver l'utilisation du Z buffer. L'affichage doit donc ce réaliser de facon "manuelle" par un tri en z réalisé par le cpu. Le tri est donc très consommateur en temps de calcul car celui ci n'est pas réalisé par la carte graphique. Afin de connaître l'ordre d'affichage des rectangles, on tri les z (dépendant de la position de la caméra) de chaque objets. La profondeur locale est obtenue en ayant connaissance de la matrice de transformation M. En notant

$$M = \begin{pmatrix} M_{11} & M_{12} & M13 & M_{14} \\ M_{21} & M_{22} & M23 & M_{24} \\ M_{31} & M_{32} & M33 & M_{34} \\ M_{41} & M_{42} & M43 & M_{44} \end{pmatrix},$$

on peux acceder à la profondeur de chaque objet (dépendant de la position locale), il suffit de réaliser la multiplication matricielle sur la troisième colonne avec la position de l'objet dans le monde réelle (x_i, y_i, z_i). On obtient donc la profondeur locale

$$z_l = M_{31}\, x_i + M_{32}\, y_i + M_{33}\, z_i + M_{44}$$

Il faut donc, pour chaque objet, stocker cette valeure z_l, puis les trier dans l'ordre, et enfin les afficher. Pour avoir moins de valeures à trier, on peux effectuer ce tri (et l'affichage) uniquement pour les z_l positifs (ceux qui sont devant la caméra). De plus, on n'affichera que les billboards étant à une distance inférieur à R de la position courante.

On notera que ce tri au niveau des quads ne permettera pas de gerer l'affichage correcte dans le cas où des arbres ou des animaux se croisent. Dans ce cas, le tri point à point par quads entier n'est pas suffisant,

[3]On remercie le site www.lespaysagistes.com mettant à dispositions ces textures détaillés, gratuites et libres de droits.

il faudrait séparer chaque quad en plusieurs triangles et trier séparément ces triangles. Cepandant, ceci augmenterait considérablement le temps de calcul.

Enfin, pour mettre les quads constamment en face de la caméra, il faut connaître le vecteur correspondant à l'axe gauche/droite et à celui de haut/bas par rapport à l'observateur. Ces deux vecteurs peuvent être obtenues à partir de la matrice de transformation. En effet, les vecteurs

$$\mathbf{u}_r = \begin{pmatrix} M_{11} \\ M_{21} \\ M_{31} \end{pmatrix} \text{ et } \mathbf{u}_z = \begin{pmatrix} M_{21} \\ M_{22} \\ M_{23} \end{pmatrix}$$

correspondent respectivement aux vecteurs droite/gauche et haut/bas local.

5.1.2 Implémentation

Les billboards sont implémentées par la classe "Transparent elements". Cette classe peux initialiser les positions des quads de facon aléatoire (ou lire des positions sauvegardées à partir d'un fichier). Les quads sont donc réparties aléatoirement et sont également de taille variable. L'initialisation d'un type d'arbre par exemple est implémentée de cette facon

```
int k_billboard=0;
for(k_billboard=0;k_billboard<N[0];k_billboard++)
  {
    position[3*k_billboard+0] = (float)(rand()%PRECISION_RAND)/(PRECISION_RAND)*max_x;
    position[3*k_billboard+2] = (float)(rand()%PRECISION_RAND)/(PRECISION_RAND)*max_y;
    position[3*k_billboard+1] = ground->get_height(position[3*k_billboard+0],
position[3*k_billboard+2]);

    scale[2*k_billboard+0] = ((float)(rand()%PRECISION_RAND)/(PRECISION_RAND)+1)*0.1/2*6.5;
    scale[2*k_billboard+1] = ((float)(rand()%PRECISION_RAND)/(PRECISION_RAND)+1)*0.1/2*6.5;

    if(position[3*k_billboard+1] == WATER_LEVEL)
      k_billboard--;
  }
```

(on remarquera que l'on évitera de placer des arbres sur l'eau en relancant un tirage si jamais la hauteur renvoyée par la surface est celle de l'eau.)

On met à jour à chaque position le vecteur correspondant à l'orientation de la caméra. La matrice de transformation est obtenue par la commande:

```
glGetFloatv(GL_MODELVIEW_MATRIX,matrix);
```

On met alors à jour les vecteurs correspondants dans la classe "Transparent element" avec la méthode "set matrix".

On affiche alors un quad donné par ces 2 vecteurs. Le résultat est montré en fig. 16

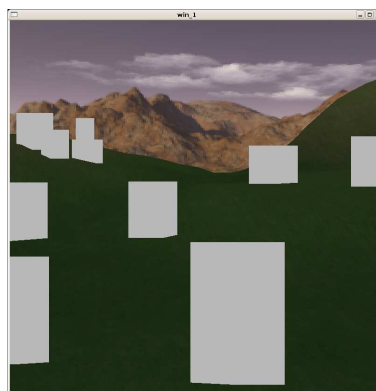


Fig. 16: Image des quads, faisant face à la caméra, placés sur le terrain.

Afin d'appliquer les textures transparentes, il est necessaire de trier la position centrale des quads en z. La méthode "sort position" de la classe "Transparent element" est appelée à chaque affichage. Pour cela, on stocke l'ensemble des z, et on trie l'index des quads en conséquence:

```
for(k=0;k<N_total;k++)
{
  new_z[k] = matrix[2]*position[3*k]+matrix[6]*position[3*k+1]+
matrix[10]*position[3*k+2]+matrix[14];

  //sort only in a defined perimeter
  if(new_z[k]<0)
   {
    d=pow(camera_position->get_x()-position[3*k],2)+
pow(camera_position->get_z()-position[3*k+2],2);
    if(d<PERIMETER_VISU_TRANSPARENT)
     {
      index_z[count]=k;
      count++;
     }
   }
}
limit=count;

//sort
ok += comparator.set_pointer(&new_z);
if(limit>0)
  sort(index_z,index_z+limit,comparator);
```

On ne trie que les objets situés devant la caméra et situés à une distance maximale de $\sqrt{PERIMETER\ VISU\ TRANSPARENT}$.

La comparaison se réalise avec la classe "Comparator" implémenté de cette facon:

```
class Comparator
{
 public:
  Comparator(float **_new_z);

  int set_pointer(float **_new_z);
  bool operator()(const int& u_0,const int& u_1);

 private:
  float **new_z;
};
```

Cette classe connait l'ensemble des profondeurs par le pointeur "new z", et le trie est implémenté par la méthode suivante:

```
bool Comparator::operator()(const int& u_0,const int& u_1)
{
  if((*new_z)[u_1]>(*new_z)[u_0])
    return 1;
  else
    return 0;
}
```

On peux alors appliquer une texture transparente sur les quads afin d'obtenir l'impression de végétation. Le résultat est affiché en fig. 17. On peux également y remarquer l'importance du tri.

Afin d'éviter un effet de répétition des textures car celle ci sont toute identiques (à la taille prêt), on inverse de facon aléatoire la texture en en prenant le symétrique par inversion des coordonnées de textures.

Fig. 17: Figure des quads avec la texture d'arbre après le trie. La première image montre un exemple de quelques arbres disséminés. La seconde et troisième figure montrent l'effet du tri. La seconde figure est en effet triée, et les arbres sont les uns derrières les autres même en grand nombre. Par contre, la troisième figure possède des arbres non triés, l'arbre au premier plan est alors mal affiché car les arbres situés derrière lui sont affichés devant.

On peux alors sans autre difficultées mettre en place d'autres textures d'arbres afin de varier le choix. Et enfin, on ajoute, avec la même technique des textures de biches. Celle ci sont donc constamment dans la même position par rapport à la caméra (à la symétrie des coordonnées de textures prêt).

Comme toujours, on peux afficher la réflexion en envoyant deux fois les quads dans le rendu. La scène prend alors rapidement une allure naturelle assez réaliste comme le montre la fig. 18

La méthode de billboard permet donc, en utilisant uniquement des objets $2D$ d'avoir une visualisation donnant l'impression d'objet en trois dimensions très complexes.

Cepandant la méthode possède quelques défauts. En effet, lorsque l'on est éloigné de l'objet, le fait que celui ci tourne avec la caméra n'est pas spéciallement visible car celui ci sort du champ rapidement. Par contre, lorsqu'un objet est proche de la caméra, celui ci tourne avec la caméra. Ce mouvement devient alors très visible et l'on peux distinguer l'astuce et le charactère $2D$ de l'objet.

5.2 Quads croisées

5.2.1 Théorie

Une autre méthode d'affichage des éléments du décor encore plus simple consiste à croiser deux quads. Ceux ci ne tournent plus avec la caméra ce qui permet d'accentuer l'effet tridimentionel de l'objet et évite l'effet obtenu lors de la rotation avec les billboards.

Cette fois, la position et l'orientation des quads sont indépendants de la positions et de l'orientation du personnage. Pour cela, il suffit d'afficher par exemple un quad parralèle à l'axe des x et l'autre perpendiculaire au premier.

On peux cependant alors voire, en tournant autour de l'objet, que l'on a des éléments $2D$ car ceux ci deviennent très fin sur les bords.

La méthode à également le désavantage de ne pas pouvoir s'appliquer sur n'importe quel type de texture. En effet, lors du croisement des quads, il faut qu'il n'y ait pas de plantes sous formé texturé sinon l'effet de croisement devient très visible. Les plantes affichées par cette méthode sont donc typiquement des plantes vertes recourbées sur les cotés.

Fig. 18: Figures des billboards des deux types d'arbres et des biches avec leur reflexion associé.

Ensuite, afin d'améliorer le dynamisme de la scène, il est facile de faire bouger la plante. Pour cela, il suffit de faire bouger légèrement la position des vertex du quad. La plante recourbée vers le bas se prête bien à ce type de déformation. On modifie pour cela le vertex central du haut de facon périodique afin d'obtenir une impression de vent.

Les textures utilisent toujours la transparence, et les quads necessitent donc la même méthode de trie. (il est nécessaire de trier les arbres et les quads ensembles dans le même vecteur sinon les quads et les arbres seront mal affichées les un par rapport aux autres).

5.2.2 Implémentation

L'implémentation des quads se réalisent également par la classe "Transparent elements". Afin de realiser l'animation, il est cepandant necessaire d'afficher les rectangles avec le point du milieu. On utilise pour cela des triangles, chaque rectangle de la plante étant découpé en 4 petits triangles. Il suffit alors de déformer le point centrale du haut en fonction du temps afin d'obtenir l'effet souhaité.

La fonction de création des triangles et de la déformation est la suivante:

```
deform_x=cos((float)kt/10-position[3*k_draw]*2*0.6-position[3*k_draw+2]*1.24*0.6)*0.05;
deform_z=cos((float)kt/9.6-position[3*k_draw]*2.4*0.6)*0.03;

glBegin(GL_TRIANGLES);
glTexCoord2f(0.0,1.0); glVertex3f(-scale[2*k_draw]/2,0.0,0);
glTexCoord2f(0.0,0.0); glVertex3f(-scale[2*k_draw]/2,scale[2*k_draw+1],0);
glTexCoord2f(0.5,0.0); glVertex3f( 0+deform_x,scale[2*k_draw+1],0+deform_z);

glTexCoord2f(0.0,1.0); glVertex3f(-scale[2*k_draw]/2,0.0,0);
glTexCoord2f(0.5,1.0); glVertex3f( 0.0,0.0,0);
glTexCoord2f(0.5,0.0); glVertex3f( 0+deform_x,scale[2*k_draw+1],0+deform_z);

glTexCoord2f(0.5,1.0); glVertex3f( 0,0.0,0);
glTexCoord2f(0.5,0.0); glVertex3f( 0+deform_x,scale[2*k_draw+1],0+deform_z);
glTexCoord2f(1.0,1.0); glVertex3f( scale[2*k_draw]/2,0.0,0);

glTexCoord2f(0.5,0.0); glVertex3f( 0+deform_x,scale[2*k_draw+1],0+deform_z);
glTexCoord2f(1.0,0.0); glVertex3f( scale[2*k_draw]/2,scale[2*k_draw+1],0);
glTexCoord2f(1.0,1.0); glVertex3f( scale[2*k_draw]/2,0.0,0);
```

```
glTexCoord2f(0.0,1.0); glVertex3f(0.0,0.0,-scale[2*k_draw]/2);
glTexCoord2f(0.0,0.0); glVertex3f(0.0,scale[2*k_draw+1],-scale[2*k_draw]/2);
glTexCoord2f(0.5,0.0); glVertex3f( 0+deform_x,scale[2*k_draw+1],0+deform_z);

glTexCoord2f(0.0,1.0); glVertex3f( 0.0,0.0,-scale[2*k_draw]/2);
glTexCoord2f(0.5,1.0); glVertex3f( 0.0,0.0,0);
glTexCoord2f(0.5,0.0); glVertex3f( 0+deform_x,scale[2*k_draw+1],0+deform_z);


glTexCoord2f(0.5,1.0); glVertex3f( 0,0.0,0);
glTexCoord2f(0.5,0.0); glVertex3f( 0+deform_x,scale[2*k_draw+1],0+deform_z);
glTexCoord2f(1.0,1.0); glVertex3f( 0.0,0.0,scale[2*k_draw]/2);

glTexCoord2f(0.5,0.0); glVertex3f( 0+deform_x,scale[2*k_draw+1],0+deform_z);
glTexCoord2f(1.0,0.0); glVertex3f( 0.0,scale[2*k_draw+1],scale[2*k_draw]/2);
glTexCoord2f(1.0,1.0); glVertex3f( 0.0,0.0,scale[2*k_draw]/2);

glEnd();
glPopMatrix();
```

Les triangles sont affichés en fig. 19.

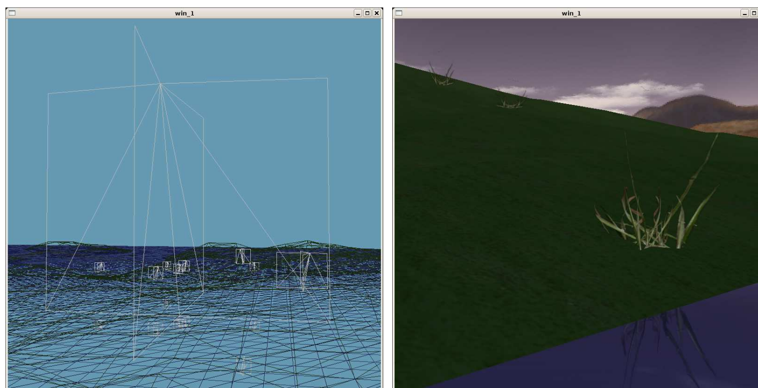


Fig. 19: Figure des rectangles croisés. La première figure est une visualisation en fil de fer où l'on peux distinguer la constitution des triangles et notamment le point centrale qui se déplace. La seconde image montre le résultat des triangles texturés.

5.3 Herbe

5.3.1 Théorie

Finalement, on peux encore ajouter une variante de rendu de végétation. Cette fois, on considère des brins d'herbes. Ces brins d'herbe doivent être très nombruex, mais par contre leur structure est simple. On peux penser à faire correspondre un brin d'herbe avec un unique triangle sur lequel on plaque la texture d'un brin comme celle de la fig. 20. Il suffit alors de définir 3 points sur la texture pour la plaquer sur un triangle. Les triangles peuvent alors être placées en grand nombre aléatoirement sur le terrain. Afin de ne pas avoir que des brins parallèles les un des autres, on peux également leur donner une orientation aléatoire.

Afin d'améliorer légèrement l'impression de $3D$, on ne se contente pas d'un unique triangle mais on découpe ce premier en 3 sous triangles, et on décale légèrement celui du haut dans le sens de la profondeur afin de donner l'impression de brin recourbé. On peux, de la même facon animer le brin d'herbe en décalant le vertex du haut pour donner l'impression de vent.

Comme l'herbe est composé d'une multitude de brins, on en affiche un nombre très important. Sur le terrain complet, on en affiche en effet plus de 200 000. Pour éviter de ralentir de facon trop importante

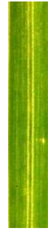


Fig. 20: Texture d'un brin d'herbe.

le rendu, on affiche uniquement les brins qui sont devant la caméra et à une distance minimale du personnage (avec la même technique que celle des billboards). Le désavantage de n'afficher que les brins les plus proche et que l'on doit parcourir une boucle sur l'ensemble des 200 000 brins à chaque affichage avec le test réalisé par le cpu. On ne peux donc pas creer de liste d'affichage une fois pour toute (même pour un grin unique car celui ci n'est pas statique). Encore une fois, un test et le déplacement pourrait être réalisé par un vertex shader afin d'améliorer la rapidité de l'affichage.

Si la représentation des brins d'herbe par cette manière est très rudimentaire et limite la géométrie du brin à une forme simple (triangulaire ou rectangulaire), elle a l'avantage de ne pas utiliser la transparence. Il n'y a donc aucun problème de Z buffer qui se charge ici de les afficher dans le bon ordre sans surcoût (le tri ici aurait un surcout très important du fait du grand nombre de brins).

5.3.2 Implémentation

L'herbe est implémenté par la classe "Grass". À chaque appel d'affichage, on envoie donc l'ensemble des vertex à la carte graphique sous cette forme:

```
//set deformation
deform_x=cos((float)kt/10-position[3*k]*2*0.6-position[3*k+2]*1.24*0.6)*0.05/5;
deform_z = cos((float)kt/9.6-position[3*k]*2.4*0.6)*0.03/5;

glPushMatrix();

//set orientation
current_cos = cos_theta[k];
current_sin = sin_theta[k];

//translate and scale
glTranslatef(position[3*k],position[3*k+1],position[3*k+2]);
glScalef(scale[0],scale[1],scale[0]);

//draw triangles
glBegin(GL_TRIANGLE_STRIP);
glTexCoord2f(0.0,1.0); glVertex3f(
  -0.003*cos_theta,
  0.00,
  -0.003*sin_theta
  );
glTexCoord2f(1.0,1.0); glVertex3f(
  +0.003*cos_theta,
  0.00,
  +0.003*sin_theta
  );
glTexCoord2f(0.2,0.3); glVertex3f(
  -0.0015*cos_theta+deform_x*0.5,
  0.015,
  -0.0015*sin_theta+deform_z*0.5
  );
```

```
glTexCoord2f(0.8,0.3); glVertex3f(
  +0.0015*cos_theta+deform_x*0.5,
  0.015,
  +0.0015*sin_theta+deform_z*0.5
  );
glTexCoord2f(0.5,0.0); glVertex3f(
  0.004*sin_theta+deform_x,
  0.02,
  0.004*cos_theta+deform_z
  );
glEnd();

glPopMatrix();
```

Le résultat est alors montré en fig. 21 pour un unique brin d'herbe.

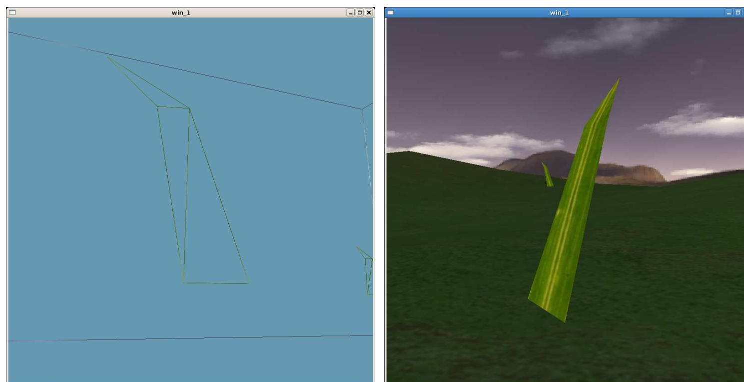


Fig. 21: Figure d'un brin d'herbe. La première image correspond au brin vu en fil de fer avec l'arrangement 3D des triangles le composant. La seconde image donne le résulat d'un brin texturé se déformant au cours du temps.

On peux également observer le rendu lorsque l'on prend un nombre important de brins. Comme le montre la fig. 22, la présence d'un nombre important de brin d'herbe peux changer l'aspect de la scène. En effet, la première image montre de petits brins en nombre très important (1 500 000 au totale sur la carte) et donne une impression de petit gazon. Au contraire, la seconde image montre un nombre un peu plus restreind de brins (400 000 au total), mais de taille beaucoup plus importante. On a alors l'impression d'un tamis d'herbes haute. Pour ces images, on peux alors aisément mélanger les billbords,

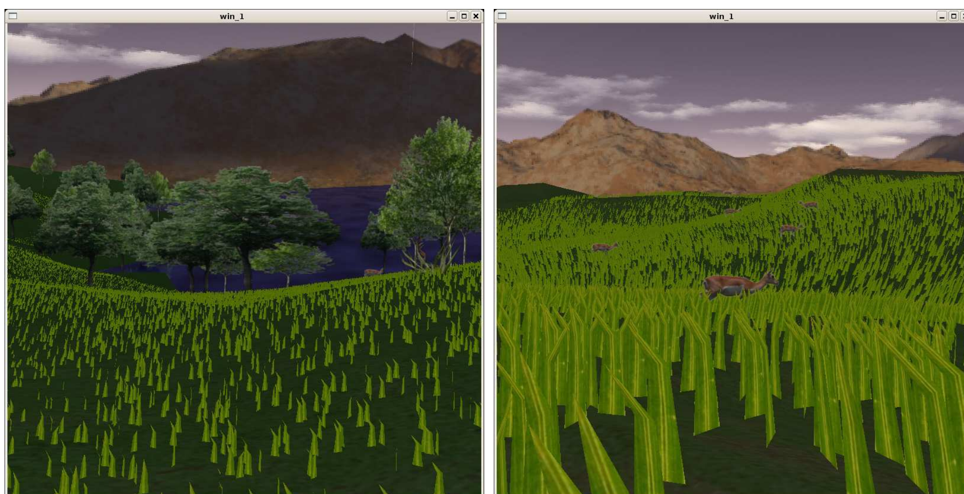


Fig. 22: Figure de l'application de l'herbe sur le terrain. La première image montre un nombre important de petits brins alors que la seconde image est formée par un nombre plus faible de brins mais de taille plus importante.

les quads croisés et les brins d'herbe afin de donner l'impression d'une nature réaliste.

6 Château

À ce stade, le monde possède l'ensemble des charactéristiques d'un milieu naturel. On peux alors rajouter des objets et détails à la scène afin de la rendre moins monotone.

On va donc rajouter dans un premier temps un château qui va nous servire d'excuse pour utiliser une technique d'impression d'illumination par blending.

Tout d'abord, on va construire un château à la géométrie très rudimentaire:

- quatre murs rectangulaires.
- quatres tours cylindriques peu détaillés afin de donner un aspect anguleux.

On va pouvoir également plaquer une texture de brique sur les murs ainsi que les tours (coordonnées cylindriques) afin de donner un aspect plus détaillé. Ensuite, le mur d'entré va être plus détaillé que les trois autres. Pour cela, on va rajouter une porte d'entrée. Celle ci sera simplement consituée par une texture de porte basique. La texture elle même sera entourée de transparence, cela permettera de plaquer cette texture sur le mur sans modifier la texture de brique du dessous. On va de plus rajouter 2 flambeaux de chaque coté de la porte. Pour cela, on va contruire un porte flambeau en vrai 3D (contitué d'un cylindre de base et d'un morceau de cône inversé pour la partie haute). Ce vrai flambeau peut facillement être texturé, car on connait sa paramétrization. On peux rapeller brièvement les équations paramétriques du cône tronqué de rayon min r_{min} et max r_{max}, et de hauteur h:

$$\begin{cases} x(u,v) = (r_{min} + (r_{max} - r_{min})\,u)\cos(v) \\ y(u,v) = u\,h \\ z(u,v) = (r_{min} + (r_{max} - r_{min})\,u)\sin(v) \end{cases}$$

Ensuite, au dessus de ces deux porte flambeaux vient se placer une flamme animée. Pour cela, on va prendre une texture de flamme (avec fond transparent) simplement plaquée sur un rectangle. La texture doit être composées de plusieurs parties montrant l'image du feu à différent moments (voir fig. 24). À chaque affichage, on va alors faire évoluer le sprite en plaquant sur le rectangle des coordonnées de texture différentes. En faisant évoluer en boucle l'animation, on obtient alors l'impression que le feux est animé. (On pourrait de plus utiliser la technique du billboarding afin d'avoir une flamme constamment orienté face à la caméra).

Finalement, on va donner une impression d'éclairement sur le mur du chateau. Pour cela, on va considérer une texture quasi transparente mais avec 2 halos jaunâtre de part et d'autres comme le montre la fig. 23. On va alors plaquer cette texture sur le mur du chateau en utilisant la technique du blending afin de mélanger les 2 halos jaunes avec la texture de briques. On va alors positionner les halos derrière les flambeaux, cela va donner une impression d'éclairement du mur due aux flammes. Ensuite, pour animer l'ensemble (afin d'éviter des flammes animées et un éclairement statique), on va animer cette texture d'éclairement en modifiant légèrement les coordonnées de textures associées aux vertex du rectangle du mur. On donnera alors l'impression d'un éclairement s'animant avec la flamme.

Les différentes textures utilisées pour la création du chateau sont montrés en fig. 23 et fig. 24

Fig. 23: Textures utilisées pour le château. La première texture correspond à la texture du base des murs et des tours que l'on plaque à répétition pour donner l'impression de briques. La seconde texture correspond à celle de la porte que l'on va plaquer sur le mur d'entré. La texture de la porte est détaillée ce qui permet de s'approcher très prêt de la porte d'entré avant de réaliser que ce n'est qu'une texture 2D plaquée. La troisième texture correspond à la texture d'éclairage due aux flammes devant le mur. C'est cette texture qui sera animée afin de donner l'impression d'un éclairage dynamique.

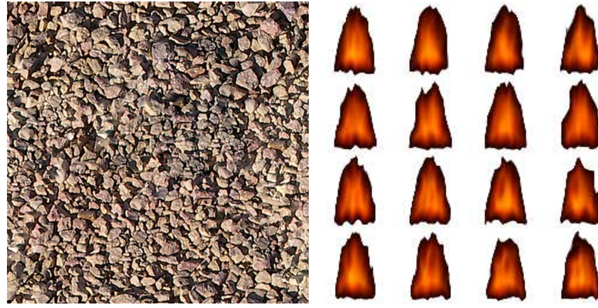


Fig. 24: Textures correspondant au feu situé devant le chateau. La première texture de type pierre est celle plaqué sur les flambeaux. La seconde texture correspond à celle de l'animation du feu. Elle permet, en ne sélectionnant qu'un quart de l'image totale pour les coordonnées de texture, de faire évoluer l'aspect du feu au cours du temps.

6.1 Implémentation

La mise en place du château se réalise avec la classe "Castle". Elle comprend trois attributs 3D fournis par la classe "Mesh tool" correspondant à une tour, au cylindre du porte flambeau et au cone du porte flambeau. Chaque élement n'est créé et sauvegardé en mémoire qu'une seule fois, les 4 tours, et 2 flambeaux sont réalisé en affichant plusieurs fois le même objet en des positions différentes. (les murs sont triviaux et donc réalisées directement "à la main")

```
class Castle
{
 public:
  Castle();
  ~Castle();

  int build_castle();
  int draw(int kt);
  int set_texture(const char* texture_path);
  int build_draw_list();

 private:

  Mesh_tool mesh_cylinder_1;
  Mesh_tool mesh_lamp_cone;
  Mesh_tool mesh_lamp_cylinder;

  GLuint texture_number[5];
  GLuint draw_list;
};
```

La mise en place du sprite du feu se réalise de la facon suivante:

```
  int k_x=0;
  float current_time = (float)kt*0.5;
  float texture_kx= (float) ( ((int)current_time)%4);
  float texture_ky= (float) (((int)(current_time/4))%4);

  //two fires
  for(k_x=0;k_x<=1;k_x++)
    {
      glPushMatrix();
      glTranslatef(0.48*k_x,0.0,0.0);
      glBegin(GL_QUADS);
      glTexCoord2f(texture_kx*0.25,texture_ky*0.25+0.25); glVertex3f(0.27-0.03,0.21,-0.07);
      glTexCoord2f(texture_kx*0.25,texture_ky*0.25); glVertex3f(0.27-0.03,0.31,-0.07);
      glTexCoord2f(texture_kx*0.25+0.25,texture_ky*0.25); glVertex3f(0.27+0.03,0.31,-0.07);
```

```
    glTexCoord2f(texture_kx*0.25+0.25,texture_ky*0.25+0.25); glVertex3f(0.27+0.03,0.21,-0.07);
    glEnd();
    glPopMatrix();
  }
```

et la déformation de l'éclairage sur le mur en fonction du temps est réalisée de la facon suivante:

```
//LIGHT

float deform_y = 0.01*powf(cos((float)kt/3.6/2),2);
float deform_x = 0.01*powf(sin((float)kt/5.4/2),2);

glPushMatrix();
glEnable(GL_TEXTURE_2D);

glEnable(GL_BLEND);
glBlendFunc (GL_ZERO, GL_SRC_COLOR);
glBindTexture(GL_TEXTURE_2D, texture_number[2]);
glColor3f(1.0,1.0,1.0);
glPushMatrix();
glTranslatef(0.0,WATER_LEVEL,0.0);
glBegin(GL_QUADS);
glTexCoord2f(0.0         ,1.0);          glVertex3f(0.0,0.0,0.0-0.002);
glTexCoord2f(0.0         ,0.0-deform_y); glVertex3f(0.0,1.0,0.0-0.002);
glTexCoord2f(1.0-deform_x,0.0-deform_y); glVertex3f(1.0,1.0,0.0-0.002);
glTexCoord2f(1.0-deform_x,1.0);          glVertex3f(1.0,0.0,0.0-0.002);
glEnd();
glPopMatrix();
glDisable(GL_BLEND);
```

La fig. 25 montre les résultats obtenues pour le chateau placé sur la petite colline du paysage.

7 Objets 3D

7.1 Théorie

Afin d'agrémenter le déplacement sur le terrain, on peux encore rajouter quelques détails. Au contraire des techniques utilisées jusqu'à présent qui était toute plus ou moins en fausse $3D$ avec plaquage de textures, on pourrait utiliser la méthode "force brute", à savoir, la mise en place directe d'un objet $3D$. Nous allons ici inclure 2 éléments de décors directement en $3D$ qui seront: un lapin et un dragon sur l'eau. Ces 2 éléments sont disponibles sur le site de Stanford sous forme d'objets .ply . Cette extension sauvegarde les formes sous la forme: tableau de sommets, tableau de connectivité.

L'objet cette fois sera réellement en $3D$ et l'on pourra tourner autour afin de l'observer. Si cette avantage peut rendre une scène beaucoup plus réaliste qu'avec les techniques utilisées précédemment, elle à aussi le désavantage de necessiter beaucoup plus de sommets à traiter que les précédentes. Il n'est alors pas rare, pour une géométrie fine, d'avoir plusieurs dizaine de milliers de points pour un seul objet. Le temps de calcul (et notamment l'envoi de la géométrie à la carte graphique) peux alors être ralenti considérablement par l'affichage d'un unique objet. Il est ainsi toujours commun de voir dans les jeux actuel, des géométrie d'objet $3D$ assez grossière avec peu de polygones mais de multiples textures plaquées, rendant la visualisation des triangles difficile. Cepandant le plaquage de texture sur la surface $3D$ n'est pas aussi facile qu'en $2D$ (pour le cas d'une géométrie dont on ne connait pas la paramétrization). Elle necessite souvent l'utilisation de texture spécifiques liées à la géométrie elle même (ex. plaquage de texture de face humaine sur un personnage).

Dans le cas présent, nous nous limiterons à l'affichage de la géométrie avec un plaquage de texture liées aux coordonnées des vertex. On créé donc un mapping de l'espace ϕ tel que $\phi : (x, y, z) \mapsto (u, v)$.

Les objets téléchargés ont des niveaux de détails importants à leur résolution maximale. Nous nous contenterons d'afficher des résolutions intermédiaires afin de ne pas ralentir de facon trop importante l'affichage.

Fig. 25: Vues du château sous différent angles. Les trois premières figures montrent la structure générale du château. Les deux première montrent ainsi la géométrie de celui ci formé par 4 murs rectangulaires et 4 tours. (on notera sur la seconde image la biche et les arbres apparaissant toujours de face même lorsque le personnage est en l'air, car le billboard fait toujours face au personnage.) La troisième image montre la structure du château vue de loin par un personnage se déplacant dans le paysage. Les trois dernières images montrent certains details de la vue de l'entré. La quatrième image montre ainsi l'arrivée vers le mur d'entré du château avec les deux flambeaux et la porte. On peux déjà également distinguer l'eclairage du mur derrière les flammes. La cinquième image est une vue de prêt de la flamme qui s'anime au cours du temps. Le porte flamme formé d'un cylindre et du cone est en vrai 3D, par contre la flamme n'est qu'un quad 2D. On peux également visualiser l'éclairement sur le mur derrière la flamme qui est lui aussi animé. Enfin la dernière image montre avec un peu plus de précision l'aspect de la porte.

Le lapin sera texturé par coordonnées cylindrique en prenant en compte ces propres vertex, et le dragon sera texturé en (x, z) afin de lui donner une impression de lignes verticales. La texture utilisé pour les deux animaux est montré en fig. 26, celle ci à été créée directement sous gimp.

Fig. 26: Texture utilisé pour le plaquage sur les animaux en 3D.

7.2 Implémentation

Le chargement des objets $3D$ est réalisé par la classe “Mesh tool” qui s'occupe du format .ply

Le lapin contient 8171 vertex pour 16301 faces, et le dragon contient 22998 vertex pour 47794 faces (à

la résolution maximale, le dragon contient plus de 400000 vertex et 850000 faces et le fichier de donnée prend à lui tout seul 30M d'espace mémoire).

On les stockent sous forme de tableau de sommets, ce qui évite la répétition des vertex pour des faces différentes et économise ainsi de la bande passante car le nombre de sommet est ici très important: le transfert de données de RAM vers la carte graphique peux donc rapidement se trouver être l'étape lente de l'affichage.

La classe "Mesh tool" se charge de stocker ces tableaux: vertex, couleur, normal, texture et connectivité. Le lapin et le dragon sont respectivement implémentés par les classes "Rabit" et "Dragon".L'affichage des données se réalise par l'appel à une liste d'affichage initialisée au début du programme (le lapin et le dragon restent fixe pendant l'execution). L'envoi des données se réalise de la facon suivante:

```
glColor4f(0.6,0.6,0.7,1.0);
glEnable(GL_TEXTURE_2D);
glBindTexture(GL_TEXTURE_2D,texture_number);

glEnableClientState(GL_VERTEX_ARRAY);
glEnableClientState(GL_NORMAL_ARRAY);
glEnableClientState(GL_TEXTURE_COORD_ARRAY);

glVertexPointer(3,GL_FLOAT,0,mesh.get_vertex());
glNormalPointer(GL_FLOAT,0,mesh.get_normal());
glTexCoordPointer(2,GL_FLOAT,0,mesh.get_texture_coordinate());

glDrawElements(GL_TRIANGLES,mesh.get_N_triangle()*3,GL_UNSIGNED_INT,mesh.get_connectivity());

glDisableClientState(GL_VERTEX_ARRAY);
glDisableClientState(GL_NORMAL_ARRAY);
glDisableClientState(GL_TEXTURE_COORD_ARRAY);

glColor4f(1.0,1.0,1.0,1.0);
glDisable(GL_TEXTURE_2D);
```

On obtient alors les objets montrés en fil de fer sur la fig. 27

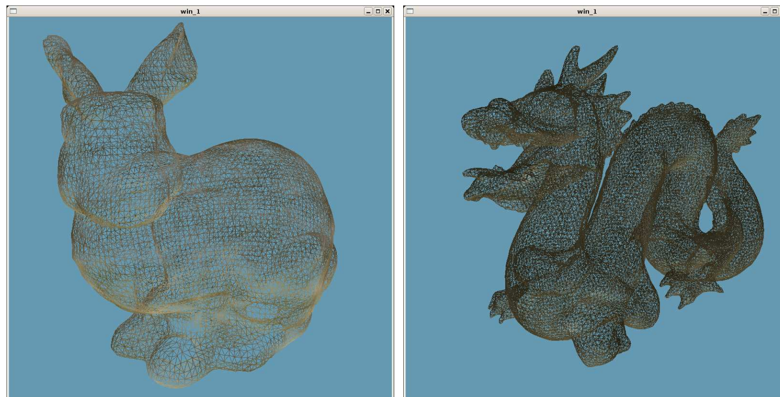


Fig. 27: Figure des objets $3D$ en fil de fer. La première image montre celle du lapin alors que la seconde est celle du dragon.

On applique alors une texture sur ces éléments. Sur le lapin, on utilise les corrdonnées cylindrique relative aux positions des sommets: Pour cela, on utilise la relation suivante (sachant que les coordonnées locales du lapin et du dragon sont centrées et comprises entre -1 et 1):

```
x = vertex[3*k_vertex+0];
y = vertex[3*k_vertex+1];
z = vertex[3*k_vertex+2];

theta = fabs(atan2(z,x))/(2*PI);
```

```
phi = (y+1)/2;

texture_coordinates[2*k_vertex+0] = theta*L1;
texture_coordinates[2*k_vertex+1] = phi*L2;
```

en notant l'utilisation de la valeur absolue sur l'angle ce qui évite le problème de raccord pour OpenGL. En effet, entre le dernier point et le premier pour l'angle, si la coordonnée de texture est à 1 alors que le point suivant est à zéro, OpenGL va interpoler sur l'espace d'un interval la texture de 1 à zéro. On va donc voir à un angle donné, l'ensemble de la texture être interpolée à l'envers sur sa totalité. Pour cela, il faut éviter cette discontinuité, l'astuce étant d'avoir la même valeure entre la première et dernière coordonnée de texture afin d'avoir un raccord correct.

Pour le dragon, les coordonnées de textures sont encore plus simple, et dépendent directement des coordonnées des sommets:

```
theta = x;
phi = z;
```

On peux alors obtenir les éléments texturés montrés en fig. 28 On peux remarquer le point centrale de la

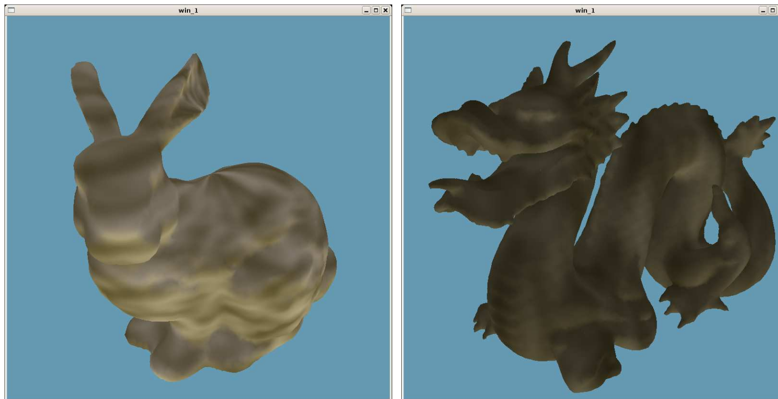


Fig. 28: Figure des éléments 2D texturés.

texture cylindrique sur le dos du lapin, qui correspond à l'axe prit pour origine.

On place alors le lapin sur le terrain, et le dragon à moitié immergé dans l'eau. On créé également la reflexion du dragon toujours avec la même technique (les sommets sont cepandant toujours envoyés 2 fois ce qui ralentirait encore plus le rendu si le dragon possède trop de vertex) afin de donner l'impression que celui ci est dans l'eau. Les résultats obtenues sont montrés en fig. 29

8 Épée et déplacement

8.1 Théorie

Afin de donner l'impression que l'on se déplace à pied, on modifie lors du déplacement la position de haut en bas. On ajoute de plus une épée se décalant légèrement lors du déplacement.

Pour donner cet effet de déplacement, on ajoute une variable globale dans le programme principale étant itéré à chaque fois que l'on avance. Cette variable *step* est alors utilisé par une fonction de type cos afin de simuler le déplacement de haut en bas.

Concernant l'épée, celle ci est un formé par un simple rectangle sur lequel on plaque la texture d'une épée comme celle montrée en fig. 30. Le reste de l'image étant remplie par transparence. Cette fois, on sait que l'image de l'épée se trouve constamment devant les autres objets du décors. La transparence ne pose donc pas de problème d'ordre d'affichage, il suffit d'afficher la texture en dernier et de ne pas tenir compte du Z buffer.

Lors du déplacement, la translation de l'épée du haut vers le bas se réalise simplement par changement des coordonnées de textures.

On implémente également la possibilitée de donner un coup d'épée. Pour cela, après l'appuie sur la touche d'action, une animation de type sprite se lance. Pour cela, on sauvegarde la texture d'épée avec

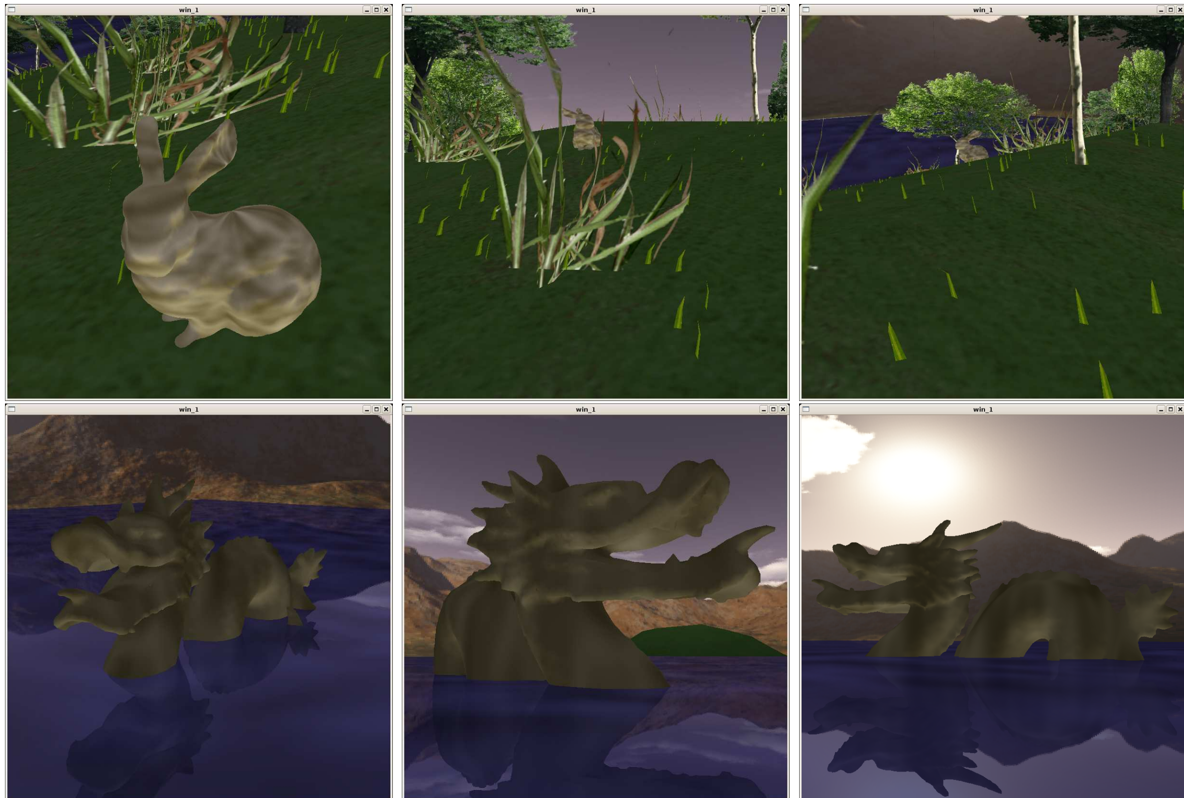


Fig. 29: Les 3 premières images sont celle du lapin. Avec une vue de prêt montrant l'effet de la texture cylindrique du le dos de clui ci, les 2 autres images sont des vues plus éloignées du lapin au milieu des éléments du décors. Les 2 images suivantes sont celles du dragon marin vu sous différents angles. On pourra observer la reflexion de la partie supérieure du dragon dans l'eau.

Fig. 30: Figure de la texture d'épée.

différentes orientations, et lors de l'animation, on plaque sur le rectangle la texture d'épée correspondant à cet instant de l'animation. Les différentes textures utilisées sont montrées en fig. 31.

8.2 Implémentation

Le déplacement du personnage se réalise par la fonction suivante dans le programme principale avec la touche 'z' étant utilisé pour avancer dans la fonction callback liée au clavier.

```
case 'z':
  current_x+=dL*sin(alpha*PI/180);
  current_z-=dL*cos(alpha*PI/180)+0.0005*cos((float)k_step/4);
  h=ground.get_height(current_x,current_z)+0.006*cos((float)k_step/3);
  position.set_position(current_x,h+0.1,current_z);
  k_step++;
```

Fig. 31: Textures de l'animation de l'épée.

L'implémentation de l'épée se réalise par la classe "Sword". En mode non animé (pas d'action avec l'épée), on utilise la fonction d'affichage suivante:

```
if(position->get_is_fight()==false)
    {
      glPushMatrix();
      glLoadIdentity();
      glTranslatef(0.0,0.005*cos((float)k_step/4),0.0);

      glDisable(GL_DEPTH_TEST);
      glEnable(GL_TEXTURE_2D);
      glEnable(GL_BLEND);
      glBlendFunc (GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
      glBindTexture(GL_TEXTURE_2D,texture_number[0]);

      glCallList(draw_list);

      glDisable(GL_BLEND);
      glDisable(GL_TEXTURE_2D);
      glEnable(GL_DEPTH_TEST);
      glPopMatrix();

      count_animation=0;

    }
```

avec le quad sauvegardé en tant que liste d'affichage. Si par contre, on active l'action de l'épée, on change alors le *glBindTexture* avec les textures correspondante:

```
glBindTexture(GL_TEXTURE_2D,texture_number[count_animation]);
```

où "count animation" est un attribut de la classe "Sword" et étant itéré par:

```
count_animation++;
if(count_animation>=4)
  {
    position->disable_fight();
    count_animation=0;
  }
```

On obtient alors l'animation d'épée montrée en fig. 32:

9 Map

9.1 Théorie

Finalement, on inclue une carte transparente du terrain vu de haut, avec la visualisation de la position courante. Cette étape nous sert d'excuse pour utiliser le viewport. En effet, après avoir affiché la scène,

Fig. 32: Animation de l'épé lors du déclanchement de l'action.

on réduit le viewport à une petite section en haut à gauche de l'image et on plaque la texture du terrain sur l'écran ainsi défini. On positionne alors un point correspondant à notre position afin de nous reprérer en temps réel.

La fenêtre de la carte est définie comme prennant le quart supérieur gauche de l'écran. On la rend également transparente par blending. De même que pour l'épée, le blending ne pose pas de problème de positionnement, il suffit de plaquer la texture en dernier sans se préocuper du Z buffer.

La carte en elle même est fixe pendant le déroulement de la scène et peux donc être sauvegardée en tant que liste d'affichage. Seule la position du point courant est alors recalculée à chaque affichage.

9.2 Implémentation

La carte est implémentée par la classe "Map". La fonction d'affichage est appelée en dernier et correspond à:

```
//set matrix to identity (no projection)
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glOrtho(-1, 1,-1, 1,0.1,2);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity ();
glViewport (0, 3*position->get_height()/4, position->get_width()/4, position->get_height()/4);

//diable Z buffer
glDisable(GL_DEPTH_TEST);

//draw Map
glCallList(draw_list);

//draw Point
glPointSize(5.0);
glBegin(GL_POINTS);
//no problem of Z buffer because GL_DEPTH_TEST is disabled
glVertex3f(-1+position->get_x()/40*2,1-position->get_z()/40*2, -1);
glEnd();

glEnable(GL_DEPTH_TEST);
```

Où l'on à pris soin de réinitialiser l'ensemble des matrices pour éviter tout type de projection (on se place en projection orthogonale). La liste d'affichage de la texture correspond à un classique affichage de quad entre -1 et 1 sur l'axe x et y avec un blending de transparence pour la texture.

Les viewports sont réglés de facon à ne prendre qu'un quart de l'écran d'affichage. En effet, on a au début de la fonction d'affichage du programme principal:

```
glViewport (0, 0, position.get_width(), position.get_height());
```

où les tailles sont mises à jour dans la fonction callback de redimensionnement:

```
static void reshape_callback (int width, int height)
{
  position.set_size(width,height);
  glViewport (0, 0, width, height);
  glutPostRedisplay ();
}
```

La classe "Position" sert à stocker cette taille de fenêtre, puis et envoyée dans la méthode draw de la classe "Map" afin de de prendre qu'un quart de la fenêtre en haut à gauche:

```
glViewport (0, 3*position->get_height()/4, position->get_width()/4, position->get_height()/4);
```

On affiche alors en fig. 33 la vue de la carte lors du déplacement dans la scène.

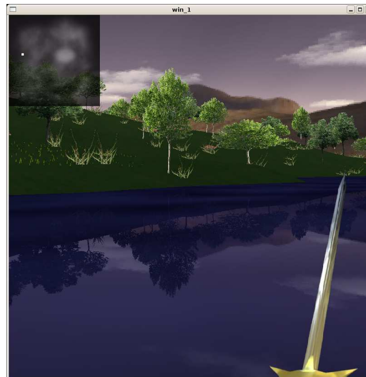


Fig. 33: Figure montrant la carte du terrain vue de haut sur le coin supérieur gauche. Le point blanc symbolise la position courante. On peux remarquer la transparence de la carte.

10 Brouillard

10.1 Théorie

Les éléments de la scène sont maintenant tous inclus. On peux essayer de donner un aspect un peu brumeux à la scène en mettant en place le fog implémenté sous OpenGL. La mise en place est très facile. Il faut cepandant faire attention à quelques points.

La mise en place d'une sky box avec du brouillard peux être problématique, en effet cette sky box est supposé être à l'infini, et ne devrait donc pas se voire. On peux cepandant la rapprocher de facon à ce que la limite du brouillard soit légèrement au dela de la sky box. Un autre problème apparaissant est que les coins de la sky box sont toujours plus éloignées de nous que le centre des murs. Ainsi, le brouillard va donc être plus épais sur les bords que sur la partie centrale des murs. On peux donc repérer suite à l'introduction du brouillard que celle ci est réalisée à partir d'un cube.

Dans notre, scène, on choisi d'implémenter un brouillard de type exponentiel carré, car cette fonction reste proche de 1 initialement puis décroît plus rapidement au delà. Ainsi la scène proche n'est que très peu brumeuse, puis le brouillard descend ensuite rapidement sur les objets plus éloignées.

10.2 Implémentation

La mise en place du brouillard est très facile, on réalise son appel une fois à l'initialisation:

```
//FOG
glEnable(GL_FOG);

glFogi  (GL_FOG_MODE     , GL_EXP2);
float fog_color[3]={0.4,0.6,0.7};
glFogfv (GL_FOG_COLOR    , fog_color);
glFogf  (GL_FOG_DENSITY  , 0.06*2);
```

Un exemple d'une scène prise dans le brouillard est alors montré en fig. 34.

Fig. 34: Figure montrant l'effet du brouillard sur une scène dégagée.

11 Discussion

11.1 Scène

On a donc pu constater que différentes techniques généralement simple à mettre en oeuvre et ne nécessitant que très peu de calcul pour le cpu permettent de donner à une scène de plein air un charactère assez naturel.

Une géométrie très simple associé à un système de texture plus développé permet donc d'obtenir des résultats visuels interessants alors que la complexité de la scène reste très limité. La scène présente a été développé et testé sur un portable équipé d'un processeur Intel 2GHz, 512M de RAM et d'une carte graphique NVIDIA Ge Force 4200 avec 64M de RAM. La scène se déroule en temps réel sans ralentissements (tant que le nombre de brins d'herbe n'exède pas la taille de la mémoire disponible).

Le terrain est déja large et possède de nombreux endroits où l'on pourraît rajouter des différents détails.

Différents points de vue dans la scène complète sont montrés en fig. 35

Fig. 35: Différentes vues de la scène reprenant certains éléments du décors. La première image correspond à une vue éloignée générale sur la végétation et la réflection ayant lieu pour le terrain, les arbres et la texture de montagne au loin. La seconde figure montre la scène lorsque l'observateur est sur le terrain. On peux y noter la présence des brins d'herbe en nombre important et la présence de biches à coté de la végétation. La mer se poursuit au loin. La troisième figure montre une vue proche du lapin au milieux de la végétation. La quatrième image donne une vue sur une biche dans un milieux d'aspect naturel. La cinqiuème image représente une vue dégagé sur la végétation avec un nombre important d'arbres. La position est choisie de facon à voir aparaîte au loin, dans la brume, le château au sommet de la colline. La sixième figure montre l'arrivé au château avec les différentes textures et le feu animé. La septième figure montre une vue à partir du sommet de la colline. On peux y voir au fond l'eau avec les petits îlots de terrains et la végétation associé. L'avant dernière figure montre la vue d'un îlot dégagé avec une biche se reflétant dans l'eau. Finalement, la dernière image montre l'approche du dragon dans l'eau avec la présence de la texture d'épée.

11.2 Problèmes

Si le déplacement dans la scène peut parraître naturelle au premier abord, une étude plus approfondie montre cepandant un certain nombre d'erreurs apparaîssant sous certaines circonstances. Certaines sont notamment intrinsèquement liées aux techniques utilisées.

11.2.1 Billboards

Les billboards peuvent rendrent des résultats impressionnant de réalisme à grande distance, cepandant ils ne sont pas sans poser quelques problèmes.

Nous avons déja évoqué le problème de la rotation. En effet, lorsque nous sommes proche de l'objet et que l'on tourne la caméra, le billboard va alors tourner avec celle ci, donnant un effet très désagréable.

La prise de hauteur peut également poser problème. En effet, comme celui ci est constamment face à la caméra, si l'on se situe au dessus du quad, celui ci sera toujours face à nous. La fig. 36 illustre ce problème lors du survol d'un arbre et d'une biche qui sont toujours face à la caméra et, dans ce cas, parallèles au sol. On pourrait régler ce problème en ne considérant la rotation qu'autour de l'axe y,

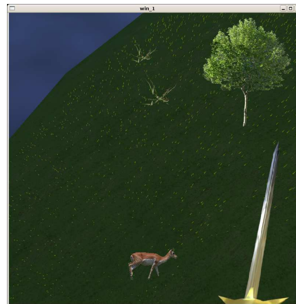


Fig. 36: Problème de billboard toujours face à la caméra même lorsqu'on le survole.

cepandant, lors de la prise de hauteur, comme le déplacement sur la petite colline par exemple, les bords des quads seraient visible et l'on perdrait plus rapidement l'impression de $3D$.

11.2.2 Quads croisées

Les quads croisées quand à eux ne posent pas de problèmes d'un point de vue de la rotation même lorsque le personnage est proche. Cepandant, la méthode possède l'inconvénient d'avoir des bordures remarquables lorque l'on se retrouve parallèle à l'axe des x ou des z.

De plus, lors de la prise de hauteur, les 2 quads sont alors clairement remarquables et forment une croix bien visible qui fait donc perdre toute impression de réalisme.

Les deux problèmes sont illustrés en fig. 37.

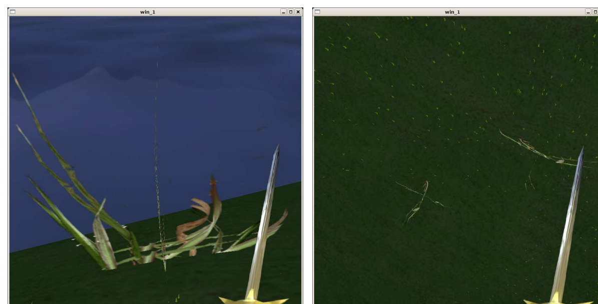


Fig. 37: Problème des quads croisées. La première image montre la vue d'une plante lorsque l'on est presque parallèle à l'un des axe: un quad de la plante apparraît comme une ligne. La seconde image montre deux plantes vue de haut qui apparaîssent comme des croix.

Ce problème est inhérent à ce type de représentation qui est trop simpliste dans les cas présenté pour donner des représentation plus correctes.

On ne détaillera évidemment pas le même problème avec les brins d'herbe. Si ceux ci sont légèrement en $3D$ due à l'inclinaison de l'un des triangle, on peux cepandant remarquer que les triangles n'ont aucune épaisseur dès que l'on se trouve sur un bord de l'un des brin. Un exemple est ainsi montré en fig. 38.

Fig. 38: Problème avec l'herbe. Le brin du milieux apparait tout plat car les triangles sont quasiment perpendiculaires à la caméra.

11.2.3 Sky Box

La sky box pose égallement quelques difficultées. En premier lieu, lors du déplacement sur le terrain, la sky box se déplace avec nous selon les axes x et z. Cepandant, on ne peux pas la faire évoluer suivant y, car sinon des détails apparaîtraient sur la partie inférieure lorsque l'on monte en hauteur, alors qu'ils étaient initialement cachées par la surface de l'eau. Cela donne alors l'impression que l'on "tire" les murs texturés de la sky box hors de l'eau.

La hauteur de la sky box est donc fixée. Cela implique donc que lorsque l'on est sur la colline, on se situe réellement plus proche des nuages, ce qui peux se voir dans certains cas. En fig. 39, on montre le cas extrème où l'on monte jusqu'à la limite de la sky box, au dela, on passe alors au dessus du plan des nuages.

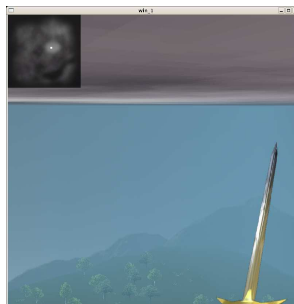


Fig. 39: Problèmes avec la sky box dont la hauteur est fixe. On peux alors se rapprocher arbitrairement des nuages.

On pourrait penser à éloigner suffisemment le ciel afin d'éliminer cet effet de rapprochement. Cepandant, cela interfère alors avec le fog qui va dans ce cas complètement recouvrir les nuages qui ne seront plus du tout visibles.

Les murs de cette sky box sont donc contraints, due à l'utilisation du fog, à rester suffisement proche du personnage. Cela pose également un problème avec les murs latéraux. En effet, ceux ci sont suffisemment proche pour couper une partie du terrain du champ de vision. Ainsi, si une colline est située derrière la sky box, on peux initialement voir le paysage texturé au loin, puis d'un coup la colline vient cacher le paysage texturé. L'effet de perspective est donc mal rendu. Un exemple est ainsi montré en fig.40. Enfin, on peux illustrer le problème déja mentionné de la distance non constante de la sky box au personnage du fait de l'utilisation d'un parallélépipède rectangle. La fig. 41 illustre l'atténuation plus importante due au brouillard dans les coins de la sky box.

Ces problèmes sont inhérent à l'utilisation combinée d'une sky box rectangulaire et du brouillard. On pourrait cepandant utiliser soit l'un soit l'autre et éviter ainsi de nombreux problèmes.

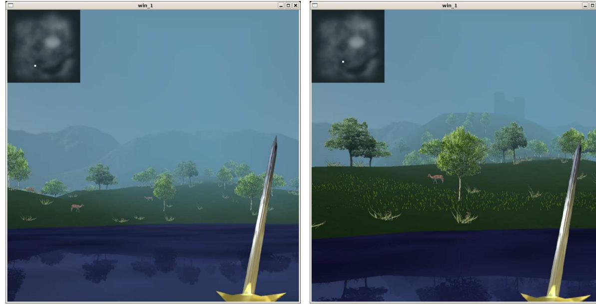


Fig. 40: Problèmes de clipping avec la sky box dont les murs sont proches. La première image montre la vue sur la sky box, la colline se situant derrière la texture de montagne située juste en face du point de vue est visible. La second image montre la même scène après avoir légèrement avancé. La texture de montagne est maintenant cachée par la colline et le château qui sont apparu de derrière le mur.

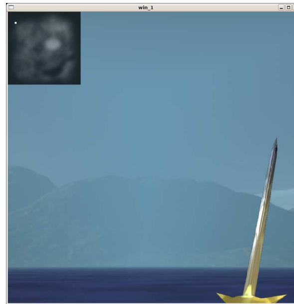


Fig. 41: Problème des coins avec l'utilisation de la sky box rectangulaire et du brouillard. Les bords sont en effet plus atténués, car ils sont situées à une distance plus grande de l'observateur.

11.2.4 Brouillard

Le brouillard pose également quelques problèmes, notamment avec l'utilisation du blending.

Dans le cas de la porte du château, on utilise trois textures, dont deux sont rajoutées en surcouche par blending. Or ces couchent, bien que transparentes, due à leur canaux alpha à 1 vont entrer en compte trois fois pour le calcul du brouillard. Chaque plan va donc atténuer la coloration du brouillard. Ainsi, la couleur de la porte principale du château se voit être beaucoup plus sombre que les tours et les autres murs l'entourant. L'illustration du phénomène est montré en fig. 42.

Fig. 42: Problème avec la porte du château et les trois couches de blending combiné à l'utilisation du brouillard.

Ce problème pourraît cepandant être réglé par l'utilisation du multitexturing avec la même méthode utilisée que pour le terrain (avec trois textures au lieu de deux). Il n'y auraît alors plus qu'une unique couche de texture et le fog serait placé correctement.

11.2.5 Textures

Enfin, si les objets en vraie $3D$ (lapin et dragon) sont géométriquement correcte sous tout les angles, la mise en place de texture ne va pas sans poser de problèmes.

La texture du lapin notamment est relativement correcte pour le contour de l'objet. Cepandant, comme la texture est cylindrique, l'ensemble des points se trouvant proche de l'axe verticale locale du lapin vont voir leur coordonnée de texture u liée à l'angle varier très brusquement. Le mapping est donc mal adapté à la géométrie du lapin. L'illustration est montrée en fig. 43

Fig. 43: Problème de texturage du lapin. Le mapping de type cylindrique pose un problème au niveau de l'axe verticale.

Ce problème n'est cepandant pas facilement rectifiable, le mapping de texture sur des objets complexe est un domaine vaste et complexe.

11.2.6 Divers points

D'autres détails de la scène et de sa gestion pourraît également être amélioré. On peux penser à la gestion de la souris. La rotation de la caméra est en effet codée par les coordonnées absolue de la souris sur l'écran. Des méthodes plus agréables pourraient être utilisées ici.

On peux également penser améliorer l'interpolation du terrain, ainsi un redécoupage du terrain par patch et interpolation plus détaillé (type spline par exemple) pourraît permettre d'avoir un terrain plus lisse. La méthode d'interpolation serait également à retravailler car certaines positions sont mal placées (légèrement en dessous ou au dessus du sol).

L'angle des billboards pourraît également être pris en compte. On pourrait ainsi forcer ceux ci à être normal à la surface, et ainsi éviter à la tête de la biche de rentrer dans le sol sous certain angles sur la pente d'une colline.

12 Conclusion

Différentes techniques ont étés utilisées avec certaines fonctionnalitées d'OpenGL. Cela a permi d'obtenir le rendu en temps réel d'une scène d'apect naturelle. Un nombre important de détails ont pu être incorporés grâce à l'aide d'un certain nombre de textures et l'utilisation de la transparence. La méthode de réflexion a, quand à elle, permi de donner l'impression de reflexion dans l'eau.

Nombre de techniques utilisées pourraient être généralement améliorées par l'utilisation des shaders. (notamment au niveau de la rapiditée de l'execution). Différentes voies sont possibles pour l'amélioration de la scène. En particulier, la mise en place de reflêts se déformant dans l'eau permetterait d'accroître le réalisme. Pour cela, on pourraît utiliser une méthode basée sur des fragments shaders permettant de déformer la visualisation du reflet. Une méthode d'émultation sur les anciennes cartes graphique pourraît également être utilisée par le chargement de la réflexion sous forme de texture, puis la déformation de celle ci en tant qu'image, et enfin plaquer cette image en tant que texture sur le plan de l'eau. Cepandant la méthode devient dans ce cas très technique comparée aux méthodes de codages directe sur la carte graphique.