PHYSICALLY BASED ANIMATION AND FAST RENDERING OF LARGE-SCALE PRAIRIE

ZHANG XIA

(*B.Eng*), ZHEJIANG UNIVERSITY

A THESIS SUBMITTED

FOR THE DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF SCHOOL OF COMPUTING

NATIONAL UNIVERSITY OF SINGAPORE

Acknowledgments

I would like to thank Associate Professor Tan Tiow Seng for providing various interesting and helpful insights on this topic. Many thanks to Rong Guo Dong, Lim Chi Wan, Ng Chu Ming, Ge Shu for sharing their invaluable experience. Thanks to Lim Chi Wan for proof-reading this report.

1 INTRODUCTION....1

1.1 RELATED WORK....1
1.2 OVERVIEW....3

2 MATHEMATICS BACKGROUND....4

2.1 THE NAVIER-STOKES EQUATIONS....4
2.2 OVERALL DESIGN....5
2.3 SOLUTION OF EQUATIONS FOR GAMES....7
2.3.1 Adding Density....7
2.3.2 Diffusion....8
2.3.3 Advection....9
2.3.4 Calculation of Density....11
2.3.5 Calculation of Velocity....11
2.3.6 Mass Conservation....12
2.3.7 Boundary Condition....15

3 NEAR PLANT DESIGN....16

3.1 MODEL OF NEAR-GRASSES....16
3.1.1 Single Blade of Grass....16
3.1.2 Grass Grid....18
3.2 ANIMATION OF NEAR-GRASSES....19
3.2.1 Cubic Bézier Curve for Wavering Grasses....19
3.2.2 Tri-linear Interpolation for Velocity....20
3.2.3 The Verlet Integration for Motion....22
3.2.4 Constraint of Grass Shape....23
3.2.5 Rotation of Grass by Wind....25
3.3 MODEL OF NEAR-FLOWERS....27
3.4 ANIMATION OF NEAR-FLOWERS....28

4 FAR PLANT DESIGN....29

4.1 BILLBOARDS OF FAR-PLANTS....29
4.2 ANIMATION OF FAR-PLANTS....33

5 LARGE-SCALE PRAIRIE DESIGN....35

5.1 OVERALL DESIGN OF SCENE....35
5.2 LEVELS-OF-DETAILS DESIGN....36
5.3 VIEW FRUSTUM CULLING....38
5.4 FAR TO NEAR SORTING....39
5.4.1 Sorting for Single Plants....40
5.4.2 Sorting for Grass Grids....42
5.5 TERRAIN DESIGN....44
5.5.1 Height Design of Terrain....45
5.5.2 Coverings of Terrain....47

6 MODEL-PLANT INTERACTION 49

6.1 EFFECT OF FADE IN AND FADE OUT 49

6.2 SPHERE-PLANT INTERACTION 51

7 RESULTS 54

8 CONCLUSION 58

BIBLIOGRAPHY 59

Summary

The realism of a real time 3D scene depends not only on the complexity of the scene, but also on the realistic animation of the objects within the scene. This project presents an integrated system for animating and rendering a real time and realistic large-scale prairie. Animating a scene requires precise calculations on the exact movement of the 3D objects. However, real time and realism are usually two conflicting objectives and thus this report also employs a series of methods for reducing computational load and yet maintains its realism. By using physical equations such as fluid dynamic equations, we are able to render the realistic animation of plants under the influence of wind movement. Grasses in the prairie are rendered either as a single blade, or as a billboard, depending on the distance of the grass from the viewpoint. Furthermore, the sequence of the grasses being rendered is also determined by using two sorting algorithm, quick sorting and pre-computed sorting. Each grass grid is uniquely rendered by varying its terrain height and the inclusion of empty patches. To improve on the performance of the rendering, view-frustum culling is used to determine the visibility of each blade of grass. In general, our output fulfills these two constraints of both real time and realism.

FIGURE 1: THE VELOCITY AND DENSITY IN THE COMPUTATIONAL CUBE.. 6
FIGURE 2: MAIN LOOP STEP FOR EQ.(1) IN THE NAVIER-STOKES EQUATIONS. 7
FIGURE 3: ADVECTION IN 2D VERSION.....10
FIGURE 4: 2D VERSION OF MASS CONSERVATION FIELD. 14
FIGURE 5: THE DESIGN OF SINGLE BLADE OF GRASS WITH 4 BACKBONE POINTS.. 17
FIGURE 6: ONE SNAPSHOT FOR GRASS GRID WITH 32X32 BLADES OF GRASS INSIDE.. 19
FIGURE 7: CONSTRAINT OF GRASS SHAPE.....24
FIGURE 8: ROTATION OF THE BLADE OF GRASS ACCORDING TO THE DIRECTION OF WIND. 26
FIGURE 9: SIX TEXTURES OF FLOWERS ARE SPRINKLED INTO THE PRAIRIE.. 27
FIGURE 10: FAR PLANTS USING BILLBOARD TECHNIQUE IN THE GRASS GRID.....32
FIGURE 11: TEXTURE OF BILLBOARD PLANTS WITH ALPHA CHANNEL.. 32
FIGURE 12: BILLBOARDS WITH TEXTURE.. 34
FIGURE 13: THE LARGE-SCALE PRAIRIE. 37
FIGURE 14: VIEW FRUSTUM CULLING.....38
FIGURE 15: TGA FORMAT USED FOR TRANSPARENT TEXTURE FILE.....40
FIGURE 16: THE PRE-COMPUTED SORTING FOR SURROUNDING GRASS GRIDS. 42
FIGURE 17: SORTING FOR A LARGE-SCALE PRAIRIE: PRE-COMPUTED SORTING. 43
FIGURE 18: HEIGHT MAP EDITOR.. 45
FIGURE 19: BI-LINEAR INTERPOLATION FOR THE HEIGHT OF ARBITRARY POSITION IN THE HEIGHT MAP OF TERRAIN. 46
FIGURE 20: FOUR DIRECTIONS OF THE TRANSFORMATION CONDITION FROM TOP VIEW.. 50
FIGURE 21: SPHERE-PLANT INTERACTION.. 53
FIGURE 22: THE AESTHETIC LARGE-SCALE PRAIRIE WITH PHYSICALLY BASED ANIMATION. 55
FIGURE 23: FINAL RENDERING RESULTS FOR DIFFERENT VIEWS WHEN THE HEIGHT OF CAMERA INCREASES. 57

1 Introduction

Grasses are commonly used in virtual world systems such as simulators and 3D gaming. In the past, most applications usually render a simple polygonal model to represent grass patches. These grass patches however, are usually motionless that provide no interaction with the wind and other objects in the scene. Our objective in this project is to provide a realistic grass rendering system for a large-scale prairie which allows us to generate not only grass-object interaction, but also for animating grass motion using physically-based dynamics.

1.1 Related Work

Grass-object interaction was introduced by IO Interactive™ in the game "Hitman Codename 47" in 2001. Although real time interaction was achieved, there was no animation of grass motion in the presence of wind. The benchmarking application of "Rendering Countless Blades of Waving Grass" in GPU Gems I [1] shows an aesthetic meadow in a valley, with grasses lying around the lake and wavering in the wind. However, no grass-object interaction is possible due to the fact that their grasses are all drawn as billboards. Furthermore, the animation of their grasses is not realistic as all the grasses are animated similarly using simple trigonometric function.

Bakay et al. [2] present a simple method for real time rendering fields of grass which are able to waver in the presence of wind. Vertex shader is used to render displacement maps with transparent shells. Their scene consists of different shells

of grasses, which are animated based on a vector that represents the motion of the wind. However, the animation is convincing only when the viewpoint is far from the animating grasses.

Shinya et al. [3] simulate the animation of trees and grass which are subjected to the complex wind fields. Their contribution is modeling the stochastic properties of wind by implementing the simple fluid flow model in [4]. However, their physical approach is considered to be too time consuming for applying onto a large-scale prairie.

Perbet et al. [5] propose animating the prairies in real time, which is similar to our objective. However, their approach is fundamentally different from ours. In their implementation, the dynamics of grass motion is handled in a much simpler approach, similar to [4]. A time-varying stochastic component is included, which allows the wind effect to be placed in an orthographic direction with respect to the terrain. The direction of the animating grass are pre-computed and stored as indexes of postures. These postures represent the direction of grass with respect to the presence of wind. Similar to [2], their animation is only realistic when viewed from afar, such as the viewpoint from an aircraft which is flying over the prairie. Nevertheless, the drawback of such a design is that the choice of stochastic wind is not based on physically accurate simulation model and the animation is restricted to limited pre-computed postures.

1.2 Overview

The objective of this project is to present an integrated system for rendering a real time large-scale prairie with physically-based animation. Section 2 introduces the Navier-Stokes equations and their solutions for each step of our implementation. Section 3 presents the design and animation for a single blade of grass and flower. Section 4 describes the design and animation for billboard grasses and flowers. Section 5 details the design of the large-scale prairie which relies on various algorithms related to the Levels-of-Details, view-frustum culling, and two kinds of sorting. The grass-object interaction is presented in Section 6. Finally, Section 7 shows the results while Section 8 is the conclusion.

2 Mathematics Background

This Chapter introduces the fluid dynamic equations for controlling the motion of wavering grass. The equations are called the Navier-Stokes equations which are used for incompressible fluid whose density and temperature are nearly constant.

2.1 The Navier-Stokes Equations

In 18th and 19th century, Claude Louis Marie Henri Navier (1785-1836) and Sir George Gabriel Stokes (1819-1903) developed a precise mathematical model for the incompressible fluid, which is known as the Navier-Stokes equations [6]. The mathematical equations are:

$$\frac{\partial \rho}{\partial t} = -(u \bullet \nabla)\rho + k\nabla^2 \rho + S \qquad (1)$$

$$\frac{\partial u}{\partial t} = -(u \bullet \nabla)u + v\nabla^2 u + f \qquad (2)$$

where ρ = density of particles such as dust in the fluid

u = velocity of fluid
t = time step
k = density viscosity of the fluid
v = kinematical viscosity of the fluid
S = density of particles injected into the computational grid
f = external velocity injected into the computational grid
$\bullet$ = dot product between vectors

∇ = vector of spatial partial derivatives, $(\frac{\partial}{\partial x}, \frac{\partial}{\partial y}, \frac{\partial}{\partial z})$ in 3D

$\nabla^2 = \nabla \bullet \nabla$

Eq.(1) is imposed for the calculation of the change of fluid density in infinitesimal time and Eq.(2) is imposed for the calculation of the change of fluid velocity in infinitesimal time.

The proposed Navier-Stokes equations provide accurate animation traits for the incompressible fluid. Although solving the equations by hand remains difficult, computers are able to calculate the equation accurately, which is very crucial for the scientific purposes. Thus, we are able to utilize the equations to model simulations such as the passage of air over the wings of an aircraft. However, even for computers, modeling the equations accurately is still too time-consuming.

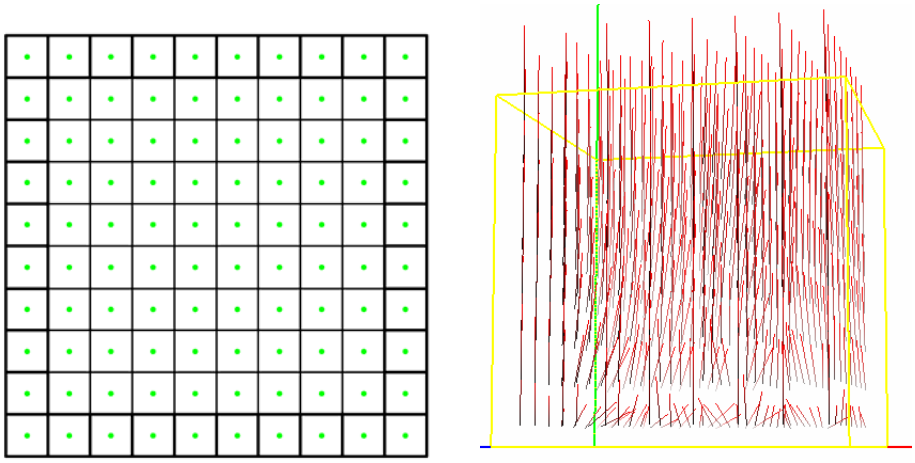
On the other hand, in gaming industry, users are less demanding on accuracy while focusing more on real time simulation and convincing animations. For this purpose, Stam [8] propose a set of new solutions for the Navier-Stokes equations which is faster but slightly inaccurate. We briefly outline these steps that lead to the calculations of the Navier-Stokes equations for real time applications.

In the next section, we discuss how the Navier-Stokes equations are applied to a 3D scene and provide pseudo code for calculating the equations based on Stam's approach [8] to a 2D scene.

2.2 Overall Design

We apply the typical Euler computational grids in computational mathematics to calculate the Navier-Stokes equations. It is a large cube (known as "*computational cube*") which contains many uniform grid cells. The variables of fluid density and velocity are set in center of each grid cell. An extra layer (known as "*boundary wall*") which also consists of similar sized grid cells covers the computational cube in order to account for boundary condition. Thus, if the cube has N^3 grid

cells, we need to allocate $(N+2)^3$ in the 3D scene. In our project, we set the value of N to be 8, thus the computational cube contains 1000 grid cells which includes the boundary walls. Figure 1.a shows one example, where the grids with lighter lines represent 8x8 cell grids. The grids with darker lines represent boundary condition to restrict the variable's motion. The dot represents the velocity and density which reside in center of each grid cell.



(a) (b)

Figure 1: The velocity and density in the computational cube. (a) Looking from the top view. (b) One snapshot in the 3D scene. Each small line represents the velocity direction in current time.

In the initial time, the fluid velocity and density are set to zero for each small grid cell, except the one called "*source grid*" which is used for injecting fluid velocity and density into the simulation. In each time step, we inject new random fluid velocity and density value into the source grid and apply the Navier-Stokes equations to calculate the diffusion and advection in each grid cell. After several time steps (it needs time to diffuse the variable of velocity and density), the velocities in each grid cell are able to obtain new values, which are used for controlling the animation of the blades of grasses. Figure 1.b shows one snapshot of velocity (shown as vector lines) for each small grid cell. The outline lines

represent bounding box of the computational grid, and the direction of each small line represents the direction of velocity in each central of grid cell.

2.3 Solution of Equations for Games

To solve the Navier-Stokes equations, we need to derive the solution to Eq.(1) that the particles move with a fixed velocity and use the solution to assist in solving Eq.(2). The rationale is due to the fact that Eq.(1) is easier to solve as it is possible to express a linear equation. A loop calculation approach, as shown in Figure 2, is used to solve Eq.(1). To simplify the notation, S is used to represent the addition of new density value into the computational cube, $k\nabla^2\rho$ represents diffusion, and $-(u\bullet\nabla)\rho$ represents advection.

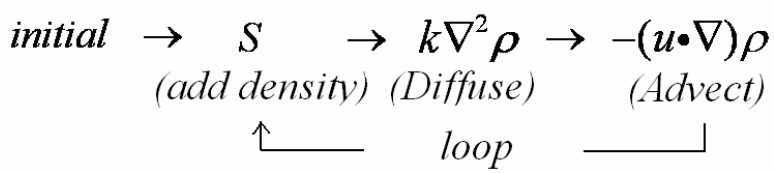


Figure 2: Main loop step for Eq.(1) in the Navier-Stokes equations.

2.3.1 Adding Density

A "*source grid*" is defined as a particular grid cell where it is selected as a source where variables are injected into the system. In each time step, a new random density value is injected into the source grid, while the densities of other grid cells are calculated by Eq.(1) in the Navier-Stokes equations.

2.3.2 Diffusion

Diffusion is the fluid phenomena which describes the motion of particles among grid cells with different particle densities within the fluid itself. Such particles move freely from one location to another. Since our implementation of fluid motion is enclosed within grid cells, we are mostly concerned with the flow of particle densities from one grid cell to another. In our 3D scene, we assume each cell can only exchange particles with its six connected neighbors. Thus for each cell, we need to calculate six terms for the flow of particles outwards and six terms for the flow of particles inwards. The mathematical equation stated below is used to solve this process.

$$x_{t+1}[i,j,k] = x_t[i,j,k] + a*(x_t[i-1,j,k] + x_t[i,j-1,k] + x_t[i,j,k-1] + x_t[i+1,j,k] + x_t[i,j+1,k] + x_t[i,j,k+1] - 6*x_t[i,j,k]) \quad (3)$$

where $x_t[i,j,k]$ represent the density of the particle present in cell $[i,j,k]$ at time t. The factor a represents the diffusion rate of the fluid. Eq.(3) has a simple structure, however, it can work only when the time step is restricted to the condition $\Delta t < \frac{l}{u}$, where l is the size of small grid cell in the computational grid and u is the motion of speed. If the time step is larger than this condition, the density in one grid cell may transmit into the non-neighbored cells such that the non-neighbored cells obtain wrong results but the corresponding neighbor cell has no contribution in this time step. So Eq.(3) is unstable and eventually the result will be blown up. Thus, to achieve the stable result for any size of time step, we change the format

of Eq.(3) to one that calculates the $x[i,j,k]$ diffused backward in time step as shown in Eq.(4).

$$x_t[i,j,k] = x_{t+1}[i,j,k] - a*(x_{t+1}[i-1,j,k] + x_{t+1}[i,j-1,k] + x_{t+1}[i,j,k-1] + x_{t+1}[i+1,j,k] + x_{t+1}[i,j+1,k] + x_{t+1}[i,j,k+1] - 6*x_{t+1}[i,j,k]) \quad (4)$$

We can build a matrix to solve Eq.(4) using a standard inverse matrix routine. But since the matrix is sparse as most items are zero, we can use a much simpler solution, "*Gauss-Seidel relaxation*" [9], to iteratively converge the result of the right side of Eq.(4). The pseudo code shows as follows:

```
// Gauss-Seidel relaxation
void linear_solver ( int N, int b, float * x, float * x0, float visc_a, float dt_t )
{
    for ( int iterate=0 ; iterate<20 ; iterate++ ) {  // iterate: numbers of iterations
        FOR_EACH_CELL
            x(i,j,k) = (x0(i,j,k) +visc_a*(x(i-1,j,k) +x(i+1,j,k) +x(i,j-1,k) +
            x(i,j+1,k) +x(i,j,k-1) +x(i,j,k+1)))/dt_t;
        END_FOR
        set_bound ( N, b, x ); }
}
// A simpler iteration technique to invert the matrix, which is called Gauss-Seidel relaxation
void diffuse ( int N, int b, float * x, float * x0, float diff, float dt )
{
    float a=dt*diff*LENGTH*HEIGHT*DEPTH;
    linear_solver ( N, b, x, x0, a, 1+6*a );  // using Gauss-Seidel relaxation
}
```

The advantage of this revision is that it would not be affected by a large time step while at the same time remain as an easily solvable equation.

2.3.3 Advection

Advection causes particles in fluid to move along the velocity direction at their position. Suppose we simplify the particle density in each grid cell into only a single particle residing in the center of the grid cell. In the first attempt, we can

calculate the new position of particle in one time step according to the velocity where the particle moves forward from the location of time t_1 to the location of time t_2 in Figure 3. However, this method can cause the same problem which is unstable if the time step is larger than the condition as we discussed in Section 2.3.2, thus the larger time step can cause the result unstable, while the smaller time step can increase the heavy load of calculation within the same period.

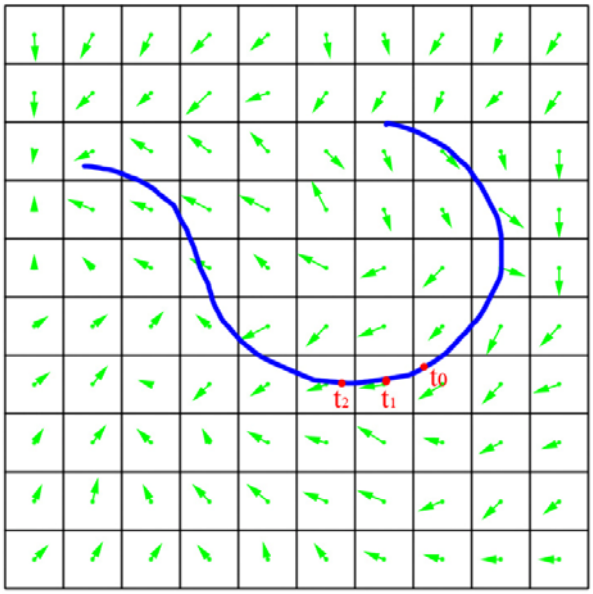


Figure 3: Advection in 2D version. Blue curve represents the particle's trajectory along time step. Green arrow represents the velocity in each grid cell. Remember the velocity can not move outside of the computational grid. Red dot represents the location of particle in time t_0, t_1, t_2. The location of particle in t_1 stays in the center of grid cell.

We can use the idea in Section 2.3.2 that inverses the direction of calculation in order to obtain the position one time step backward, such that the particle moves back from the location of time t_1 to the location of time t_0 in Figure 3. Suppose the density's position moves to the center of grid cell in time t_1, while in time t_0 ($t_0 = t_1 - \Delta t$), it resides in the position that the density can move to the position of t_1 after one time step forward. The amount of density in the position in time t_0 can

be calculated by tri-linear interpolation based on the densities of six connected neighbors in time t_0, and assigned to the density in time t_1.

2.3.4 Calculation of Density

We can combine above three steps, adding force, diffusion, and advection, together to form the calculation of density. The code is show as follows:

```
// N: number of grid; *x: density in current step; *x0: density in previous step;
// *u: x axis speed; *v: y axis speed; *w: z axis speed; diff: diffuse parameters;
// dt: time step.
void dens_step ( int N, float * x, float * x0, float * u, float * v, float * w, float diff,
float dt ){
    add_source ( N, x, x0, dt );
    SWAP ( x0, x ); diffuse ( N, 0, x, x0, diff, dt );
    SWAP ( x0, x ); advect ( N, 0, x, x0, u, v, w, dt );
}
```

where SWAP(x0,x) is the macro that exchanges data in two arrays, it is defined as follows:

```
#define SWAP(x0,x) { float * tmp=x0; x0=x; x=tmp; }
```

2.3.5 Calculation of Velocity

We can use similar steps in calculating densities to calculate the velocities in Eq.(2). The three terms in Eq.(2) are that, f represents adding velocity; $v\nabla^2 u$ represents viscous diffusion; $-(u\bullet\nabla)u$ represents self-advection which states that the velocity field itself is moveable. The pseudo code of calculating velocity shows as follows:

```
// velocity step calculation,
// N: grid size; *u: x axis speed; *v: y axis speed; *w: z axis speed;*u0: x axis old speed;
// *v0: y axis old speed; *w0: z axis old speed
// visc: viscosity parameter; dt; delta t, time step
```

```
void vel_step ( int N, float * u, float * v, float * w, float * u0, float * v0, float * w0,
float visc, float dt ){
    // add force
    add_source ( N, u, u0, dt ); add_source ( N, v, v0, dt ); add_source ( N, w, w0, dt );

    // viscous diffusion
    SWAP ( u0, u ); diffuse ( N, 1, u, u0, visc, dt );          // 1: x axis direction
    SWAP ( v0, v ); diffuse ( N, 2, v, v0, visc, dt );          // 2: y axis direction
    SWAP ( w0, w ); diffuse ( N, 3, w, w0, visc, dt );          // 3: z axis direction

    project ( N, u, v, w, u0, v0 );                          // mass conserving

    SWAP ( u0, u ); SWAP ( v0, v ); SWAP ( w0, w );

    // self-advection
    advect ( N, 1, u, u0, u0, v0, w0, dt );
    advect ( N, 2, v, v0, u0, v0, w0, dt );
    advect ( N, 3, w, w0, u0, v0, w0, dt );

    project ( N, u, v, w, u0, v0 );                          // mass conserving
}
```

Compared to the functions for the steps of calculating density in Section 2.3.4, the difference here is that the steps in velocity calculations introduce a new routine called `project()` which is not presented in the density step. It is an important aspect in the calculation of velocity. In the next section, we demonstrate the function of `project()`.

2.3.6 Mass Conservation

There is still one step we have to solve before we have finished the velocity calculations, and that is the mass conservation of the fluid. The mass conservation of the fluid represents that the fluid that flows into a cell should be equal to the fluid that flows out of this cell. However, in practice after calculation of velocity

steps without the function of `project()` this is not the case. In this section, we need to correct the situation in this final step.

Without the mass conservation, the velocity calculation will usually result in an un-natural fluid animation which contains many vectors pointing either all inward or all outward (the second item on right hand side in Figure 4.a), while in nature the fluid is a swirling-like flow (the first item on right hand side in Figure 4.a). To correct this un-natural fluid animation, we refer to a mathematic theory called "*Helmholtz-Hodge decomposition*" [11], which defined as that, each vector field (in our example, the one shown on the left hand side in Figure 4.a) is the sum of a mass conservation field (the first item shown on the right hand side in Figure 4.a) and a gradient filed (the second item shown on the left hand side in Figure 4.a). The mass conservation field looks like a beautifully swirling-like flow; on the other hand, the gradient field is the worst case for simulating the fluid since it represents the direction of steepest descent of the velocity in the fluid. Thus, the objective of the `project()` function is used for the mass conservation of velocity field, which remove the gradient field from the current result. By using the gradient field to subtract from the current vector field, we are able to obtain the mass conservation field (Figure 4.b).

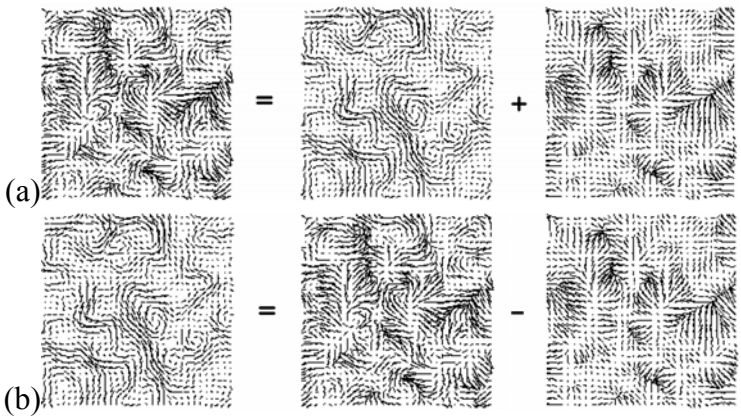


Figure 4: The steps to calculate the mass conservation field in the velocity steps in 2D version. (a) Left side: the result prior to calling `project()` routine; on right side, the first item is a mass conservation field; the second item is a gradient field. (b) The sequence that we calculate the mass conservation field. (Courtesy from Stam [8])

The gradient field can be solved by a linear system called "*Poisson equation*" [12], a second-order partial differential equation commonly used in physics calculations. Since this linear system is sparse symmetrical with most items zero, for our project, we can re-use the method of Gauss-Seidel relaxation as discussed in the diffusion steps to solve it. The pseudo code for `project()` routine is shown as follows:

```
// project, result is mass conserving field.
void project ( int N, float * u, float * v, float * w, float * p, float * div ){
    FOR_EACH_CELL
        temp = -0.5f*(  ( u(i+1,j,k) - u(i-1,j,k) ) / gridCellLengthofX +
                        ( v(i,j+1,k) - v(i,j-1,k) ) / gridCellLengthofY +
                        ( w(i,j,k+1) - w(i,j,k-1) ) / gridCellLengthofz  );
        div[IX(i,j,k)] = temp;
        p[IX(i,j,k)] = 0;
    END_FOR
    // Gauss-Seidel relaxation
    linear_solver ( N, 0, p, div, 1, 6 );
    FOR_EACH_CELL
         u[IX(i,j,k)] -= 0.5f*(p(i+1,j,k)-p(i-1,j,k)) / gridCellLengthofX;
         v[IX(i,j,k)] -= 0.5f*(p(i,j+1,k)-p(i,j-1,k)) / gridCellLengthofY;
         w[IX(i,j,k)] -= 0.5f*(p(i,j,k+1)-p(i,j,k-1)) / gridCellLengthofZ;
    END_FOR
    set_bound ( N, 1, u ); set_bound ( N, 2, v ); set_bound ( N, 3, w );
}
```

2.3.7 Boundary Condition

The boundary condition is used for restricting the calculations of velocities and densities values inside of the computational cube. In our 3D scene, the function `set_bound()` sets the velocity along *X*, *Y*, *Z* axis to zero on the boundary wall perpendicular to the respective axis. However, we can also set different rules for the boundary conditions such as allowing the velocity vectors to diffuse from one end to the other or allowing the velocity vectors to reverse their direction upon reaching the boundary walls.

3 Near Plant Design

A large-scale prairie includes grasses and flowers which can waver and rotate in the presence of wind. Depending on the distance of the grasses and flowers from the viewpoint, there are different implementations for rendering them. There are two implementation designs, one for viewing close-up while the other is for viewing far-distance.

In this chapter, we describe the rendering of grasses and flowers when viewing close-up. We define "*near-grasses*" to be grasses that are close to the camera, while "*far-grasses*" to be grasses that are far distance from the camera. We define "*near-flowers*" and "*far-flowers*" similarly. In Chapter 5, we describe the judging conditions of "near" and "far" from the camera position in more details.

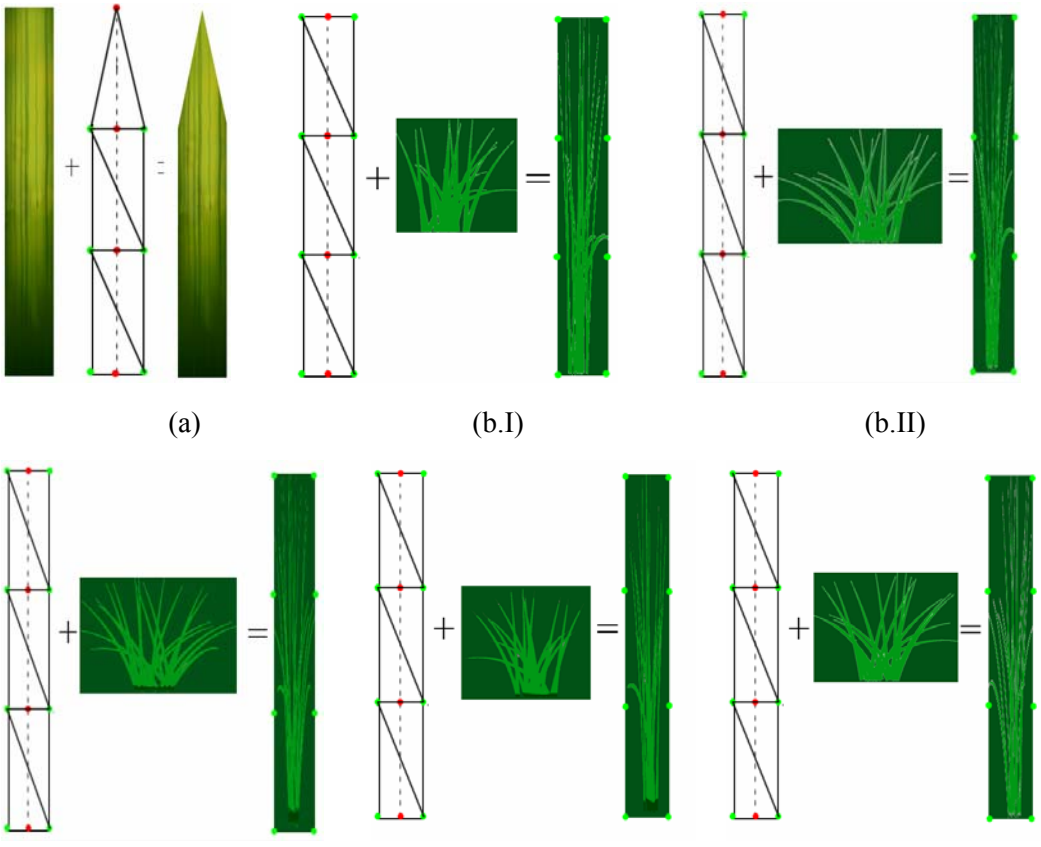
3.1 Model of Near-Grasses

For near-grasses, we first consider a single blade of grass and describe its design structure.

3.1.1 Single Blade of Grass

To represent the motion of near-grasses, we use four "*control points*" for each single blade of grass. The four control points of each blade of grass are subjected to motion vectors which will govern the animation of the blade. At each time interval, a Bezier curve is calculated from the control points. The calculated Bezier curve is then denoted as the "*backbone*" of the single blade of grass. A

group of "*backbone points*", which lies on equal intervals on the backbone, is used to control the resolution of the blade of grass. Each backbone point then extends sideward in both directions, which are always perpendicular to the upward direction, to form two "*segment points*". Thus, for any two adjacent backbone points, we have four segment points. These four segment points are used to form two triangular polygons for one segment of blade. With more backbone points, the blade of grass appears smoother, but it requires more processing time.



(a) (b.I) (b.II)

(b.III) (b.IV) (b.V)

Figure 5: The design of single blade of grass with 4 backbone points. Middle red points mean backbone points, outside green points mean segment points, triangles combine to form grass polygon, and dashed lines is backbone line of grass. (a) Single blade of grass with simple texture. (b.I~b.V) Single blade of grass with complex texture.

In our implementation, we present two different rendering models for a single blade of grass with different textures. In Figure 5.a, four backbone points are

determined from the backbone of the grass. Three of the backbone points are used to form two rectangular polygons, while the last backbone point is used to form a triangular tip. A texture of a single blade of grass is then placed onto the two rectangular polygons and one triangular polygon. In the other rendering model, we construct three rectangular polygons from the four backbone points. For the texture however, we use complex textures with alpha channel as shown in Figure 5.b (I~V) for aesthetic reason. Note that we have implemented the complex textures in this project, but for our explanation in this report, we may use the simpler version.

3.1.2 Grass Grid

In order to render the large-scale prairie, we divide it into smaller and simpler portions, known as the "*grass grid*" which is the same size as the computational grid in Section 2.2.

Within a single grass grid, we place *N*x*N* blades into one grass grid with *N* blades along the length of the grass grid and *N* blades along the width of the grass grid. To further improve on the randomness of the placement of the grasses, we use a small random location offset and an angle rotation offset for each blade of grass along the upward direction. Each blade of grass is set to the same height as the height of the grass grid with a little offset so that each control point in a blade of grass can be animated by the velocity vectors in the computational grid. Figure 6 shows a snapshot of one grass grid.

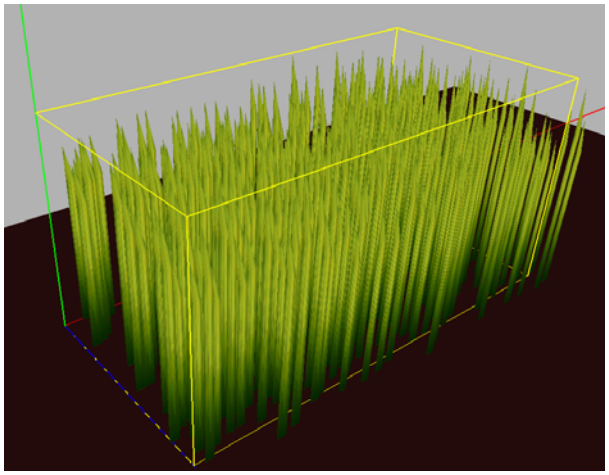


Figure 6: One snapshot for grass grid with 32x32 blades of grass inside. Yellow lines surround the grass grid.

3.2 Animation of Near-Grasses

The most important aspect in our project is to simulate the animation of grasses as natural as we see in the real world. In this section, we demonstrate how to achieve the animation. To show realistic result, each model and animation for blade of grass contains five components of design: the cubic Bézier curve for the wavering of grass model, the tri-linear interpolation for the velocity of grass animation, the Verlet integration for motion distance of grass animation, the shape constraints for the model of grass, and the rotations of grass model in the presence of wind effect.

3.2.1 Cubic Bézier Curve for Wavering Grasses

To simulate the wavering of grasses in the wind, we employ the use of cubic Bézier curves [13]. The choice of cubic Bézier curve is based on the following observations. In the presence of wind, a wavering blade of grass is usually restricted to a small angle tilt from its static position. This motion of tilting differs

from the motion of wavering long hair as hair tends to swirl around itself. Thus, it is unnecessary to employ techniques such as the kinematics technique used in hair animation. Furthermore, it is appropriate to use the cubic Bézier curve for animating grass motion as it is an aesthetically-curved shape.

The control points discussed in Section 3.1.1 are used to control the cubic Bézier Curve. Furthermore, in Section 3.2.2, the control points act as vertices which are allowed to move along the direction of wind motion. In order to calculate a backbone point lying on the cubic Bézier curve, we use the following equation:

$$p = P_0(1-t)^3 + 3P_1t(1-t)^2 + 3P_2t^2(1-t) + P_3t^3 \quad (5)$$

In Eq.(5), p is a backbone point in the blade of grass, P_i, (i=0, 1, 2, 3) are four control points of the blade of grass. Note that only P_0 and P_3 lie on the actual curve, while P_1 and P_2 provide the vectors to control the curvature of the cubic Bézier curve. t is the parameter value between 0 and 1 such as 0, $\frac{1}{n-1}$, $\frac{2}{n-1}$, …, 1, where n is the number of backbone points. By varying the value of t between 0 and 1, we can determine the backbone points lying on the curve. Thus, to find the middle three backbone points on the curve when n=5, we set the value of t to 0.25, 0.5, 0.75. The more backbone points we set on the curve, the smoother the curve is, and the more computational time will be consumed.

3.2.2 Tri-linear Interpolation for Velocity

In this section, we discuss how the force is calculated which acts on the control points of the grass. In Chapter 2, we demonstrate how to calculate the velocity

vector for each grid cell in the computational grid. However, we only compute the velocity vector in the center of each grid cell. Thus, we need a method to compute the velocity vector at the position of the control point since it does not usually locate in the center of the grid cell.

Firstly, we determine the grid cell which the control point resides in. Secondly, we determine the corner of the grid cell which the control point is closest to. Thirdly, we locate the eight grid cells which are incident to that corner. Using the velocity vectors at the centre of the eight grid cells, we apply the tri-linear interpolation to calculate the velocity vector for the control point. We show the pseudo code as follows:

```
// set velocity to grass, tri-linear interpolation used now
void Grass::SetVelocity(float *uu, float *vv, float *ww, int xGrid, int yGrid, int zGrid,
                        float gridLength){
    FOR_EACH_GRASS_BLADE
     FOR_EACH_GRASS_CONTROL_POINT
        // get the position of current control point in grass grid
        float x = m_grassBlade[i]->GetParticlePos(j)->x / gridLength;
        float y = m_grassBlade[i]->GetParticlePos(j)->y / gridLength;
        float z = m_grassBlade[i]->GetParticlePos(j)->z / gridLength;

        // calculate the grid cell with minimum x,y,z of current particle
        if (x<0.0f) x=0.0f;  if(x>xGrid) x = (float)xGrid; i0=(int)x;    i1=i0+1;
        if (y<0.0f) y=0.0f;  if(y>yGrid) y = (float)yGrid; j0=(int)y;    j1=j0+1;
        if (z<0.0f) z=0.0f;  if(z>zGrid) z = (float)zGrid; k0=(int)z;    k1=k0+1;

        // calculate the distance between particle and cell with minimum x/y/z length
        r1 = x-i0; r0 = 1-r1;  s1 = y-j0; s0 = 1-s1;  t1 = z-k0; t0 = 1-t1;
        // check particle close to which cell, and set the calculation grid to this cell
        if( r1>=r0 )  i0 = i1; if( s1>=s0 )  j0 = j1; if( t1>=t0 )  k0 = k1;
        // get correct 8 grids for tri-linear interpolation
        i1 = i0;  i0 -= 1;   j1 = j0;    j0 -= 1;    k1 = k0;   k0 -= 1;

        // apply tri-linear interpolation
        CVector3 temp;
```

```
            temp.x =      r0*s0*t0*uu[i0,j0,k0]+r0*s1*t0*uu[i0,j1,k0]+
                          r1*s0*t0*uu[i1,j0,k0]+r1*s1*t0*uu[i1,j1,k0]+
                          r0*s0*t1*uu[i0,j0,k1]+r0*s1*t1*uu[i0,j1,k1]+
                          r1*s0*t1*uu[i1,j0,k1]+r1*s1*t1*uu[i1,j1,k1];

            temp.y =      r0*s0*t0*vv[i0,j0,k0]+r0*s1*t0*vv[i0,j1,k0]+
                          r1*s0*t0*vv[i1,j0,k0]+r1*s1*t0*vv[i1,j1,k0]+
                          r0*s0*t1*vv[i0,j0,k1]+r0*s1*t1*vv[i0,j1,k1]+
                          r1*s0*t1*vv[i1,j0,k1]+r1*s1*t1*vv[i1,j1,k1];

            temp.z =      r0*s0*t0*ww[i0,j0,k0]+r0*s1*t0*ww[i0,j1,k0]+
                          r1*s0*t0*ww[i1,j0,k0]+r1*s1*t0*ww[i1,j1,k0]+
                          r0*s0*t1*ww[i0,j0,k1]+r0*s1*t1*ww[i0,j1,k1]+
                          r1*s0*t1*ww[i1,j0,k1]+r1*s1*t1*ww[i1,j1,k1];

            m_grassBlade[BLADE_NUM]->SetForce( CONTROL_NUM, temp );
        END_FOR
    END_FOR
}
```

3.2.3 The Verlet Integration for Motion

The next component of animation of near-grasses is to define the calculation of velocity both for the purpose of simplicity and stability. In each time step, the distances traveled by control points with respect to the wind force are calculated. The control point has two variables: the position x and the velocity v. After one time step, the position will travel to x' and the velocity will change to v'. We can easily get the equation before and after one time step.

$$\begin{aligned} x' &= x + v \cdot \Delta t \\ v' &= v + a \cdot \Delta t \end{aligned} \qquad (6)$$

where Δt is the time step, and a is the acceleration. This is a simple Euler integration. The drawback of Eq.(6) is that it is not accurate since the magnitude of error introduced by the Euler integration is in $O(\Delta t^3)$ [10].

In our project, we use a more accurate and faster representation: instead of storing each control point's position and velocity, we store its current position x and previous position x''. The equation is then:

$$\begin{aligned} x' &= 2x - x'' + a \cdot \Delta t^2 \\ x'' &= x \end{aligned} \qquad (7)$$

This is called the Verlet integration [14] and is widely used for simulating molecular dynamics [15]. The magnitude of error introduced by the Verlet integration is in $O(\Delta t^4)$ [14] which makes the Verlet integration an order more accurate than the Euler integration. The pseudo code is shown as follows:

```
// the Verlet integration
void GrassBlade::Verlet(float time){
    for(int i=1; i<4; i++) {
        // back up the old Position
        CVector3 temp = m_controlPos[i];

        // the Verlet integration
        m_controlPos[i] = 2*m_controlPos[i] - m_controlOldPos[i] + m_velocity[i] * time;
        m_controlPos[i].y = abs( m_controlPos[i].y );  // constrain positive on y axis

        // x”=x
        m_controlOldPos[i] = temp;
    }
}
```

3.2.4 Constraint of Grass Shape

Without any constraints, each backbone point in the blade of grass can move along the direction of wind freely. The shape and the length of grass are lost and the result becomes un-realistic. In this section, we add the constraint of grass shape, no matter how the backbone points in grass blade animate, the shape of grass can be controlled.

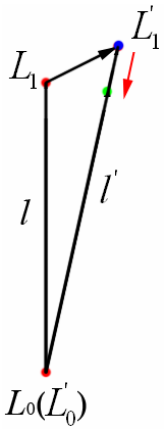


Figure 7: Constraint of grass shape. The red dot represent the positions of a vertex initially; the blue dot represents the position of vertex after one time step; the green dot represents the position of vertex under the constraint condition; the red arrows represent the trajectory of vertex moving backward under the constraint condition.

In the initial time, suppose L_0~L_3 are the locations of backbone points P_0~P_3 in one blade of grass (see Figure 7). The distance of two adjacent backbone points has the same length l, that is, $|L_1 - L_0| = |L_2 - L_1| = |L_3 - L_2| = l$, where $L_1 - L_0$ is a vector and l is a scalar. We first analyze the calculation of the segment P_0P_1 which contains the backbone point P_0 residing on the ground. P_1 moves to the new location L_1' according to the presence of wind, $L_0\,(L_0')$ is the same since it represents the position of root of grass and should be immovable. The length of the segment changes to $|L_1' - L_0| = l'$, which is not equal to l in most case. Since the length of offset vector $L_1' - L_1$ is a tiny scalar compared to the length of the segment l, we can safely set the distance traveled $|L_1' - L_1|$ to $l' - l$ and move backward with the vector of $\frac{l' - l}{l'}(L_1' - L_0)$ (the red arrow in Figure 7). The other segments are calculated by the above analysis from the sequence of bottom to top of the grass. The code is shown as follows:

```
// constraint the grass' particle moving
void GrassBlade::Constraint (int index){
     float initSegmentLength = m_heightBound / numberOfSegment;
     CVector3 delta = m_backbonePos[index] - m_backbonePos[index-1];
     float length = delta.Magnitude();  // the length of delta

     // the percentage beyond the length of segment
     float diff = (length- initSegmentLength) / length;

     // subtract the beyond length
     m_backbonePos[index]   -= delta * diff;
}
```

3.2.5 Rotation of Grass by Wind

Through the observation of real grasses, we realize that the blade of grass will rotate with respect to the direction of wind. To simulate this animation, we calculate the rotation to set the blade of grass be perpendicular to the direction of wind. In this section, we show how the function of rotation works for the blade of grass.

To rotate the blade of grass, two segment points which are extended from the same backbone point are rotated along the upward direction in the middle point (the backbone point). The degree of rotation is calculated by the following assumption.

We assume that two segment points extended from the same backbone point will rotate along the upward direction instead of along the direction of two adjacent backbone points as in the real world. This assumption changes the length of blade since the line connecting two segment points is not perpendicular to the line

connecting two adjacent backbone points, however, it simplifies the calculation of rotation and the result remains aesthetically pleasing.

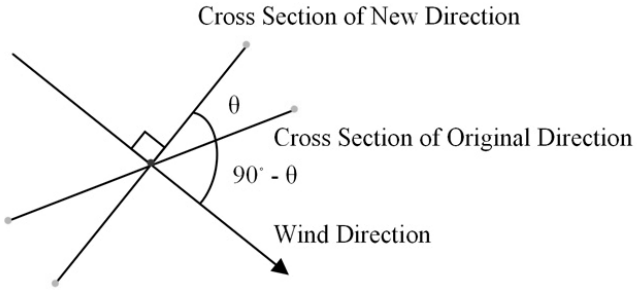


Figure 8: Rotation of the blade of grass according to the direction of wind, from top view. The dots at the end of lines represent segment points; θ is the degree of rotation; the dot in the middle of line represents the backbone point.

The calculation of rotation is as follows (in Figure 8). Firstly, the "*wind direction*" is calculated by the direction of vector from the bottom of backbone point on the ground to the top of the backbone point in the air which represents moving towards the direction of wind. Secondly, the "*blade direction*" is obtained from the vector of two bottom segment points which represent the initial position of the blade of grass on the ground. Thirdly, the dot product of normalized vector of wind direction and normalized vector of blade direction is cos(θ) so it is easy to calculate θ by calling acos(θ) function. Finally, we rotate two segment points evenly with degree $\frac{\theta}{n}i$ along the upward direction in each segment of a blade so that the rotation looks smoothly from bottom to top, where n is the number of segments, and i is the index of the current segment of grass and $i \in [1, n]$.

3.3 Model of Near-Flowers

It seems a little boring if only grasses exist in the prairie. Thus, to enrich the scene, we sprinkle some flowers into the prairie in this project. From the observation in the real world, the stem of the flowers are usually covered by the brushwood and only the petals are visible, so it's not necessary to draw the flower model including stalk and foliage.

In our project, the model of near-flowers is a simple square which represents the petal of flowers. The square of petal is perpendicular to the upward direction. A little random offset along the upward direction for each flower is set so all flowers do not point to the same direction uniformly. In Figure 9, we set the textures of flowers from the real photos. In the initial period, we randomly select one from these six textures and assign it to the model of flower.

Figure 9: Six textures of flowers are sprinkled into the prairie. All textures are selected from the real photos.

To position the near-flowers, we randomly sprinkle the flowers in one grass grid. The rendering method is such that, we concurrently check the condition for allocating flowers when rendering a single blade of grass, and if the condition of flowers is satisfied then we plant them in the corresponding position. To achieve

the effect that flowers are hidden in the blades of grass, we set the height of petals to 0.65 times of the height of blade with a little random offset.

3.4 Animation of Near-Flowers

The animation of near-flowers employs the same idea of the animation of blade: the direction of wind controls the animation of flowers. One difference between flowers and grasses is that, we only need to calculate the animating direction of petals which is similar to the top point of backbone of grass and no need to calculate the animations of other backbone points in the manner that are done in a blade of grass.

Another difference in the animation between flowers and grasses is that, the petal of flower and the blade of grass have different animating speed. In the real world, the stalk of flower is more rigid than the blade of grass so that the flower animates with a little slower speed and a shorter distance than the grass under the same wind. To achieve this effect, we add a new variable, the *mass*, into the class of grass and flower and set the rule as follows: the heavier the mass, the less the model can animate in the wind. When applying the velocity to the position which the petal resides in, we multiply velocity to a number in proportion to the inverse of mass. For example, we can set the mass of vertex on the ground to be infinite and the reciprocal is 0 such that the velocity of vertex on the ground is 0, thus the vertex is immovable.

4 Far Plant Design

It is not necessary to render the grasses and flowers which are distant away from the camera position using the same technique as the near-grasses. There are two main reasons for it. The first reason is that, for objects distant away from the camera position, the users are not able to identify a single wavering grasses and flowers among the group. In this case, it is more attractive to animate a group of plants on a whole than individually for each grass or flower object. The second reason is that, according to current hardware capacity, it is impossible to achieve real time rendering if we render all grasses and flowers individually in a large-scale prairie.

In this chapter, we describe how to design the grasses and flowers distant away from the camera position. Since we do not use different technique for grasses and flowers, we call them "*plants*" together instead of grasses and flowers, respectively.

4.1 Billboards of Far-Plants

For the grass grid which is distant away from the camera position, we have the following observation: the farther the distance between the camera position and the grass grid, the smaller the image of plants projected on the screen, and thus a smaller number of polygons that we need to render. It is obvious to choose the technique of billboards to render a group of far-plants. The benefit of using

billboards is to achieve similar visual effect and real time rendering by employing a fewer polygons for models rather than rendering the models individually.

To achieve the visual effect that the billboards of far-plants appearing the same as rendering a group of single plants, in our project, we combine the single plants to form billboards in one grass grid with the following rules:

- Each billboard of plants is composed of six rectangles connecting together at the middle line which is pointing to the upward direction (see Figure 10 and Figure 12).
- The length of billboard is represented by the numbers of single blades of grasses between two rectangles with the same orientation.

The length of billboard depends on the distance between the "*camera grid*" (where the camera position resides in) and the other grass grid. We set the length of billboard as Eq.(8) and the numbers of billboards in one grass grid as Eq.(9):

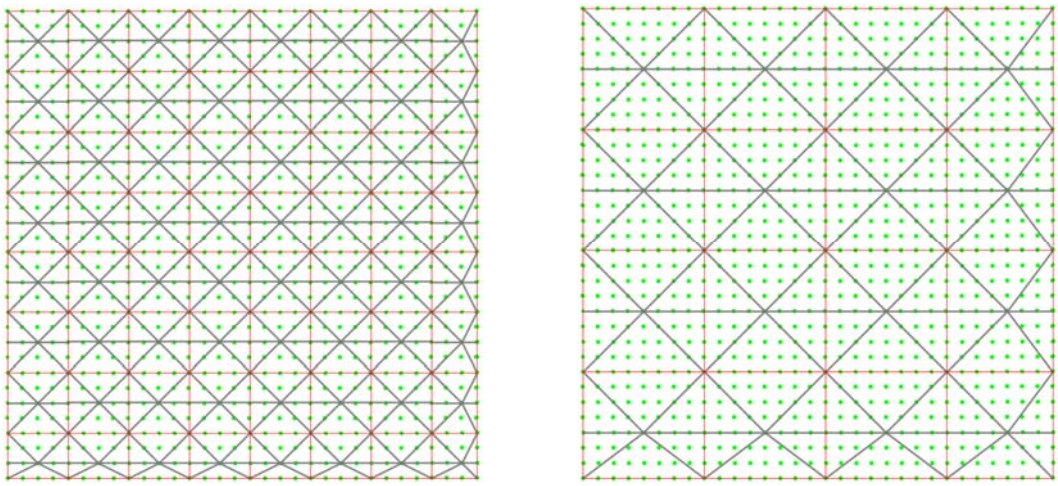
$$l' = l * 2^{\lfloor \log_2 g \rfloor} + 1 \qquad (8)$$

$$n' = \left\lceil \frac{n-1}{l'-1} \right\rceil \qquad (9)$$

where l is default length of billboard and we set l=2 in our project, l' is the length of billboard in each grass grid which is distant away from the camera position, g is the maximum number of grid offsetting along X and Z axis between the grass grid and the camera grid; n is the numbers of single plants along X and Z axis respectively in one grass grid, and n' is the numbers of billboards along X and Z axis respectively in one grass grid. Eq.(8) is easy to program in C language since we can use the bit-shift function for the logarithm and power of 2. Eq.(8)

can guarantee that the maximum number of grids away within the range from 2^{n-1} to 2^n-1 can be rendered with the same length of billboard to keep the same visual effect.

We show an example how to set the length of billboard and the numbers of billboards, and compare the numbers of polygons rendered between single plants and billboards in one grass grid. For each grass grid, suppose we have 32 blades along X and Z axis, respectively. We need to render 1024 blades of grass for a single grass grid. If each blade of grass has 3 segments, we need to render 3072 rectangles for one grass grid. By using the technique of billboards, we can reduce the numbers of polygons significantly, as compared with drawing single plants. Consider the case where the camera resides in grass grid [2,2]. The grass grid [5,4] is *max*(5-2,4-2)=3 grids away from the camera grid. Using Eq.(8), each billboard represents $l' = 2 * 2^{\lfloor \log_2 3 \rfloor} + 1 = 5$ plants, except for the last billboards which have lesser number of plants to represent (as shown in the right and bottom billboards in Figure 10). Using Eq.(9), in this grass grid, there are $\left\lceil \frac{32-1}{5-1} \right\rceil = 8$ billboards along X and Z axis, in total 64 billboards (see Figure 10.a) and we need to render 64x6=384 rectangles since each billboard has 6 rectangles combining together. Similarly, the grass grid [6, 7] is *max*(6-2,7-2)=5 grids away from the camera grid, and the billboard represents $l' = 2 * 2^{\lfloor \log_2 5 \rfloor} + 1 = 9$ single plants, there are $\left\lceil \frac{32-1}{9-1} \right\rceil = 4$ billboards along X and Z axis, in total 16 billboards (see Figure 10.b) and we only need to render 16x6=96 rectangles.



(a) (b)

Figure 10: Far plants using billboard technique in the grass grid, from the top view. The green dot points represent the single plant; the light red rectangles represent the length of one billboard group; the light blue lines represent rectangles in billboard. (a) One example of billboards with length of 5 in the grass grid. (b) One example of billboards with length of 9 in the grass grid.

In order to display the similar visual effect for the billboard in the far-distance and single plant in the close-up, we design the texture of billboards based on the single texture of blade of grass and flower shown in Figure 5 and Figure 9. Figure 11 shows the texture of billboard which is a simple rectangular picture with several blades of grasses and flowers.

Figure 11: Texture of billboard plants with alpha channel. The grasses and flowers combine together to simulate the same visual effect in the close-up.

It is the technique of billboard that helps us achieve the goal of real time rendering with the same image projection for the far-plants as rendering individually for the

single plants. For different distance of grass grid, it has different length of billboard and different number of billboards, which is called “Level-of-Details”. We re-emphasize it in the design of a large-scale scene in Section 5.2 where we introduce the large-scale prairie and put billboards and single plants into the scene.

4.2 Animation of Far-Plants

In Section 4.1, we show that the far plants are drawn as billboards with six rectangles combining together, and define the length and texture of billboards, but we do not define the position of each vertex in billboards in order to control the animation. In this section, we specify it and design how it can animate in the presence of wind.

The essential idea for the animation of billboard is the same as that for the animation of single plant. The segment points in single plants are used to form the vertices of rectangles in billboards. After calculating the length of billboards according to the distance between the camera grid and the other grass grid in Section 4.1, we connect the segment points in two single plants by offsetting the length of billboards to form the rectangular polygons. Furthermore, to simplify the computational time, only the bottom and top vertices of segment points in single plants are used to render the rectangular polygons in billboards, since it is not necessary to display the animation of middle segment points in single plants for the distance far away from the camera position.

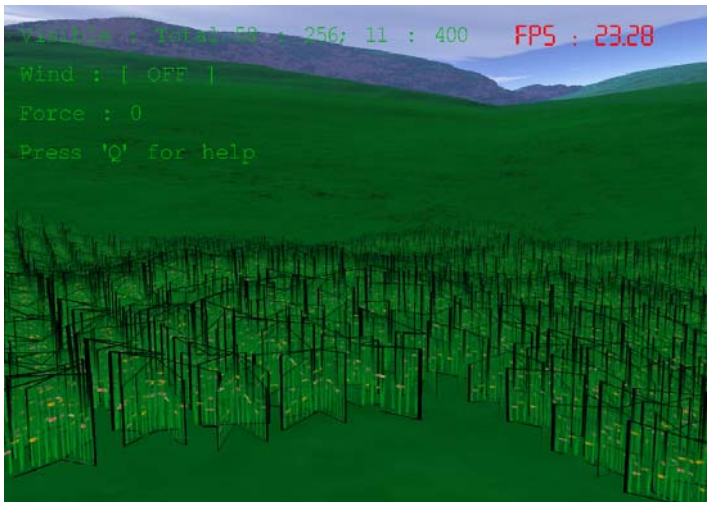


Figure 12: Billboards with texture. The darker lines of each billboard demonstrate the real polygonal shape of rectangles.

Figure 12 shows an example of billboard in our project, according to Eq.(8), the near billboards represent 5 single blades of grasses combining together while the farthest billboards represent 9 single blades of grasses, which are organized as in Figure 10 for each grass grid. When the billboards project on the screen, they have the similar visual effect although the farthest grass grids have smaller numbers of polygons than the near grass grids. When the wind direction is added to the grass grid, the vertices of billboards animate like the wavering single plants which are similar to the plants closer to the camera position.

Since the farther billboards only occupy small images when projecting on the screen, the wavering animation is not easy to be identified compared with the single plants which animate around the camera position. To display the same animating speed between far-distant billboards and close-distant single plants, we increase the velocity for the billboards in proportion to the distance between the grass grid and the camera position.

5 Large-Scale Prairie Design

In this chapter, we describe how to combine two types of models, single plants and billboards, in a large-scale prairie. We have outlined the design of prairie, which consists of many individual grass grids. The single plants and billboards are then placed into the prairie, depending on the distance between the grass grid and the camera position. In the following section, we describe the implementation in more details.

5.1 Overall Design of Scene

The setting of the scene is as follows. The world coordinate system in the scene is such that the X and Z axis are parallel to the ground, with the Y axis pointing towards the upward direction. Within the scene, we place 16x16 grass grids for simulating the large-scale prairie. In one single grass grid, 32x32 plants are placed within. To compute the Navier-Stokes equations, we divide the computational grid into 8x8x8 grid cells, plus the boundary wall of extra grids. Thus in total, we have 10x10x10 grid cells in a single computational grid. Note that the size of a grass grid is similar to the size of the computational grid.

To reduce the computational time for animating the prairie, we only apply the Navier-Stokes equations for a single grass grid, and duplicating the solutions for the other grass grids. Thus the velocity at a particular grid cell for a grass grid is similar to its counterpart grid cell in another grass grid throughout the animation. It is therefore possible that two grasses in different grass grid residing at the same

position, relative to its grass grid, will result in exactly the same animation. However, since we offset the location of each grass grid by a random value, the chances of similar animation is reduced greatly. Furthermore, we have gaps among plants, which increase the diversity of the pattern of grasses in each grass grid. Thus, even though the solutions to the Navier-Stokes equations in each grass grid are similar, the resulting animation is different.

5.2 Levels-of-Details Design

In this section, we detail how the choice of using either single plants (near grass grid) or billboard plants (far grass grid) in each grass grid is decided.

First, the grass grid (known as "*camera grid*") which the camera resides in is calculated. Together with the camera grid, eight other grass grids (known as "*surrounding grid*") which are adjacent to the camera grid, if exist (do not consider the view frustum culling in this section), are used to render single plants individually.

For the other grass grids in the prairie, we use billboard plants to represent a set of plants. The length of the billboard depends on the maximum number of grid offset from the camera grid in Section 4.1.

We show in Figure 13 to illustrate the setting of the whole scene consisting of the large-scale prairie. The figure is shown from the top view, and each small grid represents one grass grid. The bottom-left grid is set to (0,0), while the camera resides in grass grid (7,8). As described previously, the grass grids surrounding

the camera grid render single plants individually, shown as white squares in Figure 13, while the other grass grids use different lengths of billboards to represent plants. The choice of the length of the billboard is shown in Section 4.1. We use the notation of (7±x,8±y) to represent grid which (x,y) offset grids away from the camera grid, which is in grass grid (7,8). Using the (x,y) values for each grid, we can thus determine the length of the billboard to employ. Using Eq.(8) from Section 4.1, there are 64 billboards with length of 5, and 16 billboards with length of 9, and 4 billboards with length of 17 in different far-distant grass grids. In Figure 13, we mark the grass grid with different billboard length with different patterns. Thus, by using the technique of levels-of-details we can reduce the numbers of polygons to be rendered while keeping the same visual effect as compared to rendering all plants one by one for the far-distant grass grids.

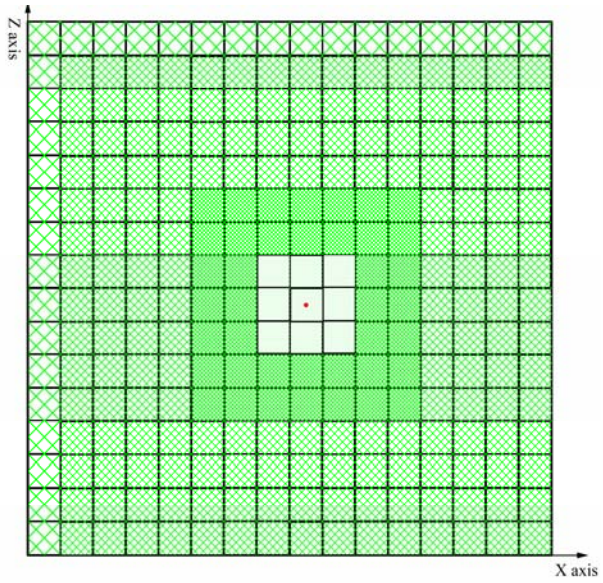


Figure 13: The large-scale prairie, from the top view. The denser the cross lines, the more numbers of billboards in one grass grid.

5.3 View Frustum Culling

When 3D object takes a promenade through the prairie, the camera can only see part of the scene because of the limitation of visual angle. Since many grass grids and plants are invisible, it is not necessary to render all of them, so we can employ the technique of view frustum culling to reduce the number of rendering objects. The basic idea of using view frustum culling is to render the visible grass grids in the scene and the visible plants in the grass grids, and skip the others.

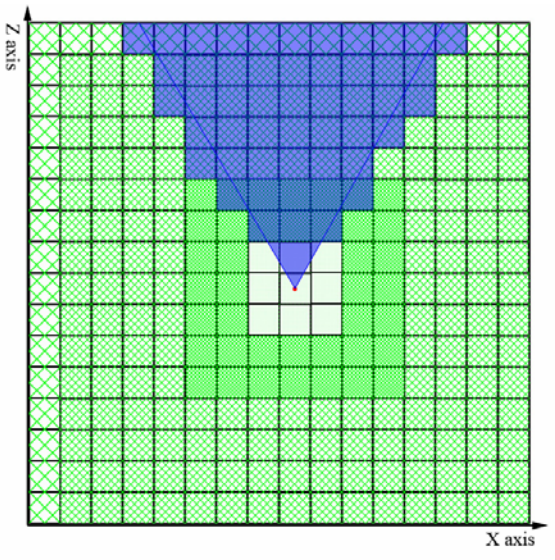


Figure 14: View frustum culling, from top view. The two light blue lines form visual angle of 60 degree. The light blue rectangles represent the visible grass grids which should be drawn in current frame. It only needs to render 50 grass grids using billboards out of total 247.

In our project, we use two levels of view frustum culling. Firstly we eliminate the invisible grass grids; secondly we eliminate the invisible single plants in the camera grid and the surrounding grids. Suppose the view angle is 60 degree, in each time step, we check the position of each grass grid with respect to the six view planes (top, bottom, left, right, near, and far plane). If the grass grid stays inside of all view planes, we set the grass grid visible; if it is outside of any one of

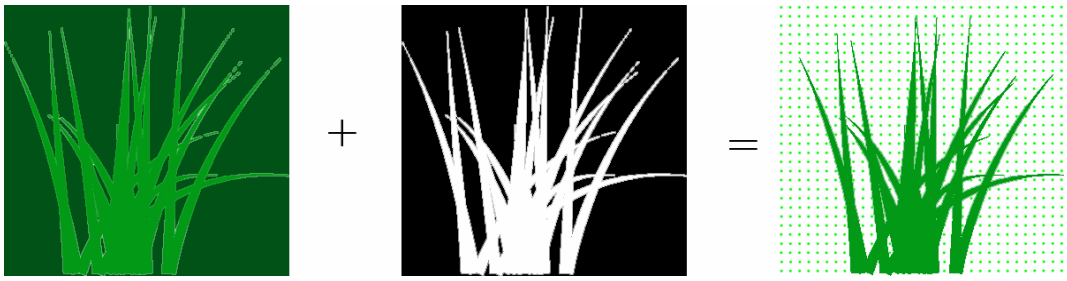
viewing planes, we ignore it. Then, we render the visible grass grids from far to near sorting using the technique in Section 5.4. Based on this setting, in each frame, we only need to render about 20 percent of all grass grids (see the number on top-left in Figure 12 and the proportion of visible grids to all grids in Figure 14), which save computational time. The single plants in the camera grid and the surrounding grids use the same six viewing planes described above to eliminate the invisible single plants. Figure 14 shows an example from the top view.

5.4 Far to Near Sorting

In our project, the complex grass texture is stored as TGA file format which contains RGB channels and an alpha channel. The alpha channel controls the transparency of the RGB channels, with 0 meaning full transparency and 255 meaning full opaque if the number is between 0 and 255. An example of texture is shown in Figure 15 which displays a blade of grass with pure black and pure white. Using transparency, we can see the grasses staying behind the blades or billboards through the components where the alpha channel is 0, which is called "*blending*" in OpenGL [16]. During blending, the color values of the incoming fragment (the Source) are combined with the color values of the corresponding currently stored pixel (the Destination) using the following equations in our project:

$$\begin{aligned} R &= S_r * S_a + D_r * (1 - S_a) \\ G &= S_g * S_a + D_g * (1 - S_a) \\ B &= S_b * S_a + D_b * (1 - S_a) \\ A &= S_a * S_a + D_a * (1 - S_a) \end{aligned} \qquad (10)$$

where R, G, B, A represent the results of blending color, S_r, S_g, S_b, S_a represent the four components of color in source, and D_r, D_g, D_b, D_a represent the four components of color in destination. Each component of R, G, B, A is eventually clamped to [0, 1].



(a) (b) (c)

Figure 15: TGA format used for transparent texture file. (a) RGB color is the same as that in BMP format. (b) Alpha channel represents the transparent and opaque part of RGB. (c) TGA format can show the translucent effect.

From Eq.(10), we acknowledge that different rendering sequences of two objects overlapping each other can affect the rendering result. The correct sequence is to first render objects distant away from the camera position, and then render the objects closer to the camera position; otherwise the rendering result looks wrong. To satisfy this request, we apply two sorting algorithms: the quick sorting and pre-computed sorting for rendering objects from far to near according to the camera position.

5.4.1 Sorting for Single Plants

We use two sorting algorithms for single plants: those that reside in the camera grid and those that reside in the surrounding grids (see Section 5.2). For the plants residing in the camera grid, we use the common method of sorting algorithm, that

is, quick sorting [17]. We build up an array of data with the distance between each plant and the camera position. In each frame, we do the following steps:

1) Determine the plants which are visible under current view angle using the technique of view frustum culling in Section 5.3.
2) Obtain each distance between visible plant and the camera position, and store the distance into an array.
3) Apply quick sorting algorithm to sort the distance from far to near.
4) Render each plant using the sequence in the sorted array.

For the plants staying in the surrounding grids, in order to save the computational time, we do not employ the quick sorting algorithm but use another strategy, which is called "*pre-computed sorting*". Pre-computed sorting is calculated initially and used during the simulation. In each frame, the correct rendering sequence for a particular grass grid from far to near according to the camera position is determined by its relative position to the camera grid. Since there are eight grids surrounding the camera grid, there are eight sequence of drawing the single plants for each type of grass grid as shown the follows.

Position	**Pre-computed Sort**	**Figure 16**
(0, +Z)	FORWARD	(a)
(0, -Z)	BACKWARD	(b)
(-X, 0)	LEFT	(c)
(+X, 0)	RIGHT	(d)
(-X, +Z)	FORWARD_LEFT	(e)
(+X, -Z)	BACKWARD_RIGHT	(f)
(+X, +Z)	FORWARD_RIGHT	(g)
(-X, -Z)	BACKWARD_LEFT	(h)

Table 1: Eight sequences of drawing the single plants in the surrounding grids.

Figure 16 shows the design of eight sequences of rendering plants for the pre-computed sorting. The direction of the arrow represents the rendering sequence in a particular row of billboard within the grass grid, while the numbers represent the sequence of the rows of billboard to be displayed. Figure 16.i represents the example of the sequence of rendering plants for surrounding grids, where the dot represents the camera position. In Figure 16.i, no matter which direction the camera views, the sequences of rendering plants in eight surrounding grids keep from far to near correctly.

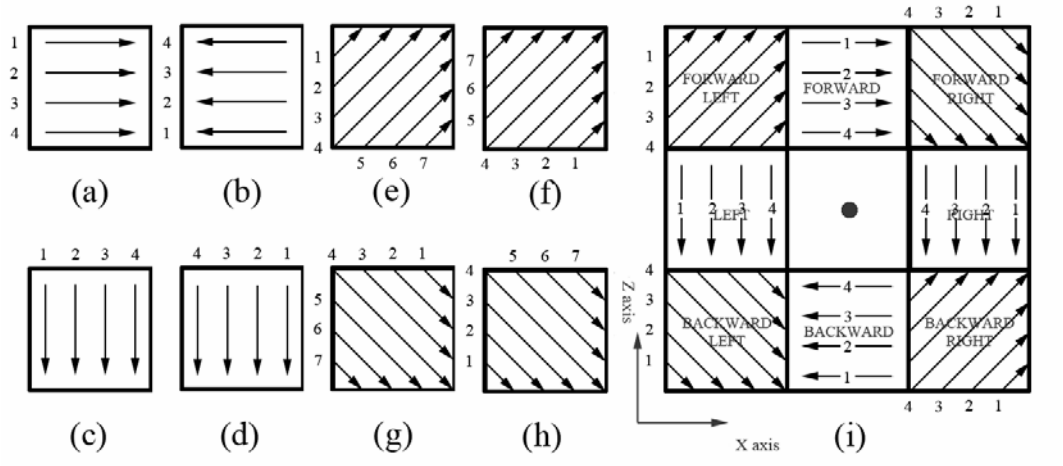


Figure 16: The pre-computed sorting for surrounding grass grids from the top view.

The benefit of using pre-computed sorting is that, we only need to calculate the sequence of rendering plants once during the initial setup, and can be easily selected during the rendering of the scene.

5.4.2 Sorting for Grass Grids

The sorting for grass grids uses both quick sorting and pre-computed sorting. The purpose of quick sorting for grass grids is to render the grids from far to near. The

purpose of pre-computed sorting is to render the billboards from far to near inside of one grass grid. We calculate the sorting in the following steps.

Firstly, we use the quick sorting algorithm to arrange the visible grass grids from far to near. The sorting priority is determined by the number of offset a grid from the camera grid. For example, in Figure 13, grass grid (1, 9) has the grid offset $(1-7)^2+(9-8)^2=37$ (square root operation is not needed, to save the computational time further) from the camera grid (7, 8).

Secondly, the sequence of rendering the billboards within a grass grid is determined by the grass grid's relative position to the camera grid. Using the same principle in Section 5.4.1, we use one of the eight pre-computed sorting sequence in Figure 16.a~h to render the correct sequence of billboards.

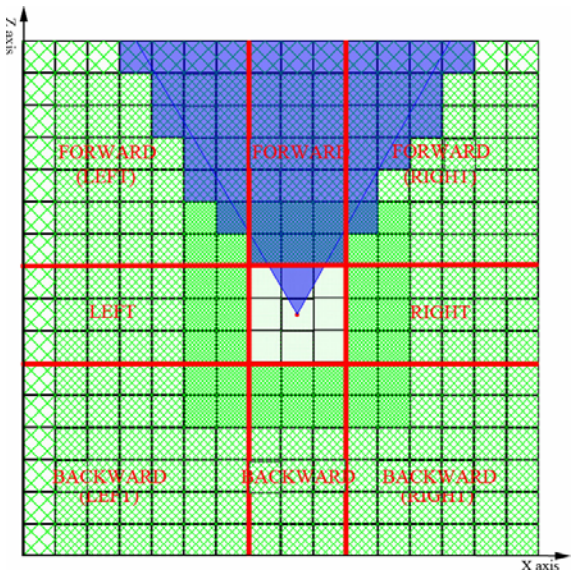


Figure 17: Sorting for a large-scale prairie: pre-computed sorting, from top view. The red lines separate the whole scene into nine sections.

We separate the whole scene into nine sections, and the grass grids in each section has the same pre-computed sorting which can guarantee the correct sequence of rendering billboards, except the middle section is used for rendering the single

plants as discussed in Section 5.4.1. Furthermore, to simplify the computation, the section of forward-left in Figure 17 can use the rendering sequence of forward or left along the direction of X or Z axis, instead of the sequence of forward-left in Figure 16.e which renders single plants along the direction of diagonal, and the same as the section of forward-right, backward-left, and backward-right.

When the camera moves to other grass grid, we only need to change the position of the camera grid, and use the same technique to solve the rendering sequence of grass grids and billboards from far to near. This judging condition will introduce an un-natural visual effect and the solution will be discussed in Section 6.1.

5.5 Terrain Design

Till now, we have set up a large-scale, but flat and orderly arranged prairie with billboards-displaying plants that are distant away from the camera position, and using single plants for plants that are close. Furthermore, the plants are rendering in a sequence from far to near with the correct visual effect. However, it is not aesthetic and convincing compared with the natural terrain, which should have hills, valleys including stochastically placed plants. In this section, we discuss the design of terrains. This section contains two components: the height of terrain and the "*covering*" of terrain defining which parts of the prairie contain plants and which parts do not.

5.5.1 Height Design of Terrain

We use a free GNU General Public License program called Height Map Editor [18] to design the height map of terrain. The height map is a 2D representation of a 3D terrain. Each pixel in the height map represents a height value which ranges between 0 and 255. Height Map Editor provides user interface to easily set and edit the height of each pixel.

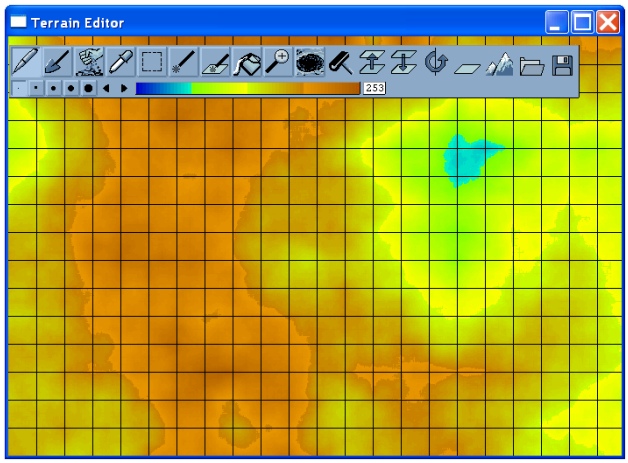


Figure 18: Height map editor. Each color represents a height which ranges from 0 to 255.

To define the resolution of the height map of terrain, suppose we have 16x16 grass grids, and each grass grid contains 32x32 plants. Therefore in total we have 512x512 plants (suppose billboards are represented by single plants). Thus the easiest way for defining the resolution is to set the resolution of height map to match the numbers of plants, that is, 512x512 pixels for the height map, thus each pixel in the height map corresponds to the height of one plant.

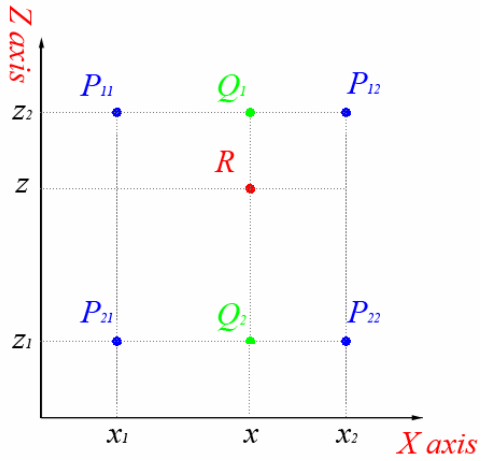


Figure 19: Bi-linear interpolation for the height of arbitrary position in the height map of terrain.

Bi-linear interpolation is used when the position of plants and 3D monster object do not match the pixel of the height map, such as in cases where the resolution of the height map does not match the total numbers of plants in the prairie, or the 3D monster takes a promenade through the prairie freely. In Figure 19, suppose four dots P_{11}, P_{12}, P_{21}, P_{22} represent the corresponding pixels in the height map, while R represents an unknown height. To calculate R, we first calculate the height of middle points Q_1 and Q_2 by linear interpolation along X axis using Eq.(11) and Eq.(12), then calculate the height of R by linear interpolation along Z axis using Eq.(13).

$$Q_1 = \frac{x - x_1}{x_2 - x_1} P_{12} + \frac{x_2 - x}{x_2 - x_1} P_{11} \qquad (11)$$

$$Q_2 = \frac{x - x_1}{x_2 - x_1} P_{22} + \frac{x_2 - x}{x_2 - x_1} P_{21} \qquad (12)$$

$$R = \frac{z - z_1}{z_2 - z_1} Q_1 + \frac{z_2 - z}{z_2 - z_1} Q_2 \qquad (13)$$

The advantage of using bilinear interpolation is that, it is easy to calculate the height of arbitrary position, even though we set the height map with a lower

resolution than the default setting that each pixel in height map matches one plant in the prairie. Another advantage is to move the 3D monster smoothly (no bump up and down suddenly) when it takes a promenade through the prairie.

5.5.2 Coverings of Terrain

In our implementation, we render all plants in the visible grass grid, which causes the scene to be too orderly and monotone, as compared with the natural environment. To increase the variety of the scene, we load other 3D models into the scene and randomize the prairie such that some parts are covered with plants while other parts are barren.

We use another height map (known as "*covering map*") to control the display of 3D models instead of the height information discussed in Section 5.5.1. For example, we set a height map with 512x512 pixels to load the objects in the prairie. Since the number of pixel is between 0 and 255, we can set a maximum of 256 different kinds of 3D models in the prairie. In our project, we use the following ID to represent 3D models:

ID	**3D Model**
0	None
253	Tree
254	Stone
255	Grass

Table 2: ID for 3D models.

In each frame, the program checks the ID in current position and renders the corresponding model. If the number in current position is 0, then nothing is rendered, which is called "*gap*" in the terrain. Each non-plant model is rendered

with a random offset for orientation, size, and rotation. For example, the stones have three orientations and rotations along XYZ axis and the trees have one orientation and rotation along Y axis.

Note that the position of gaps or other models can separate the billboard into several parts. Each part of small billboards is still a group of rectangles which have the small length compared with original billboards.

The advantage of the covering map is that, it is easy to define and edit the grass patches since we can re-employ the height map editor, and do not need to revise any existent code in our project.

6 Model-plant Interaction

In this section, we describe two aspects of interactions: the first one is to add the smooth transition that transforms the billboards to single plants or vice versa when the camera moves from one grass grid to another; the second one is to add the interaction between the 3D monster model and its surrounding plants as it walks through the prairie.

We first introduce the setting of the 3D character and the control keys for moving camera. The 3D model format is called “MD2” which is introduced by id Software™ when releasing Quake 2 in 1997, it's quite simple to understand and use. The camera stays behind the monster with a small fixed distance and looks towards the monster initially. User can press key ‘W’ and ‘S’ to move both the monster and the camera forward and backward, and press key ‘A’ and ‘D’ to rotate the orientation of the monster and the camera around the monster.

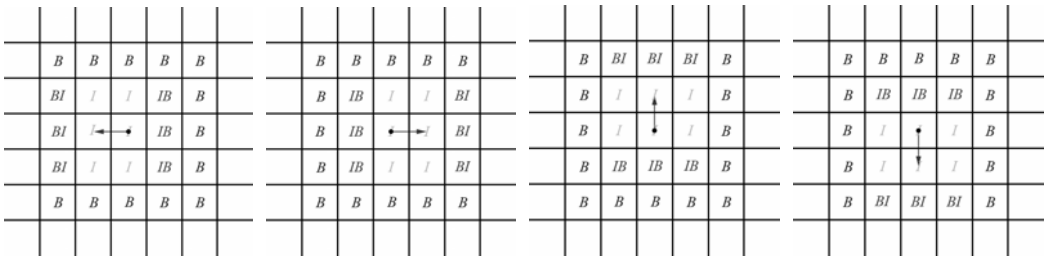
6.1 Effect of Fade in and Fade out

One artificial visual effect of the animation is that, as the camera traverses from one grass grid to another, some grass grids surrounding the camera position immediately transform from single plants to billboards while some grass grids transform from billboards to single plants. The reason is because of the judging condition for setting the billboards and single blades in our project: the grass grid which the camera resides in and eight surrounding grids (if exist) will be rendered

as single plants, while the others will be rendered as billboards. Thus, the transition is very sudden and disturbing to the viewer.

The solution to this problem is to add a function to detect which grass grid is about to transform from billboards to single plants, and draw both billboards and single plants with opposite alpha channel in this grass grid, then set a transit ratio to the alpha channel to obtain a smooth transition of fade-in and fade-out effect.

To achieve the first goal, first we determine the movement of the camera. In Figure 20, four possible movement of the camera is displayed and the grids which are going to transit from single plants to billboards ("*IB*") and from billboards to single plants ("*BI*") are shown. Using the four conditions in Figure 20, we can determine the transition state (if any) for each grass grid.



(a) (b) (c) (d)

Figure 20: Four directions of the transformation condition from top view. "*B*" represents rendering *B*illboards in the grass grid. "*I*" represents rendering single plants (*I*ndividual) in the grass grid. "*BI*" represents that the model of plants transform from *B*illboards to *I*ndividuals when the camera traverses to another grass grid and "*IB*" vice versa. The arrow represents the moving direction of the camera. Dot means current position of the camera. (a) The camera moves to LEFT. (b) The camera moves to RIGHT. (c) The camera moves to TOP. (d) The camera moves to BOTTOM.

To achieve the second goal, we set a boundary sphere (known as "*Transition Sphere*") residing in the camera position. If the transition sphere touches the boundary of a grass grid along X or Z axis, we render the grass grid twice, using the rendering methods of both billboards and single plants concurrently for visible

grass grids (*BI* or *IB*). We then vary the alpha channel of both billboards and single plants accordingly to achieve the transition.

Under the same speed of the 3D monster, the radius of the boundary sphere (which is fixed) determines the speed of transformation from billboards to single plants or vice versa. The transformation begins only when the transition sphere intersects with the boundary of the grass grid. The alpha channel of the fade-in object (either single plant or billboard) is set to $1-\frac{r_1}{r_0}$, where r_1 is the distance between the center of the transition sphere and the boundary of the grass grid and r_0 is the radius of the transition sphere. Correspondingly, the alpha channel of the fade-out object is set to $\frac{r_1}{r_0}$. Hence, the shorter the radius, the quicker the transformation is. Based on this approach, we can achieve the smooth transformation between the billboards and single plants when the camera traverses into the other grass grid.

6.2 Sphere-plant Interaction

In this section, we describe how plants are pushed aside from the monster's body and restored to the original position after the monster passes over the plants. To achieve this goal, we use the boundary sphere (known as "*Interaction Sphere*") for the monster. The calculation step is as follows.

Firstly, we set an interaction sphere residing near the position of monster's foot. The interaction sphere does not fully envelop the monster in this project since we

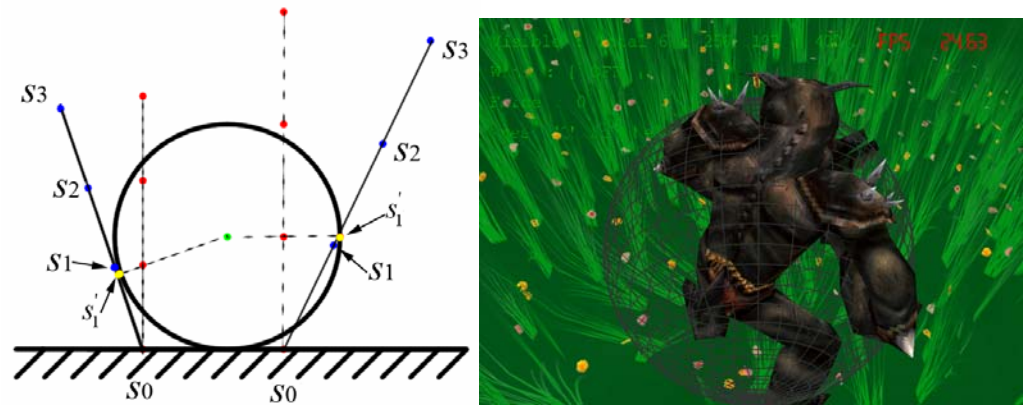
only need to consider the interaction for the low-lying plants, which reside in the position near the monster's foot. This design depends on the requirement of the application, for example, if the monster passes over the plants which are higher than the monster, we need to set the boundary sphere enveloping the monster.

Secondly, we project the interaction sphere on the ground to form a circle and check whether the root of plant resides inside of this circle, and those plants is involved in the sphere-plant interaction.

Thirdly, we detect the first backbone point staying inside of interaction sphere (for example s_1 in Figure 21.a, and we follow the sequence that s_0 is below s_1, and s_2 is above s_1) from bottom to top in a single plant, then move the backbone point outside to $s_1^{'}$ along the vector from the center of interaction sphere to the initial position of s_1.

Fourthly, we use the constraint technique in Section 3.2.4 to restrict the length of segment $s_0 s_1^{'}$ to obtain the real vertex point s_1 in the blade of grass, and calculate the vector from s_0 to s_1.

Lastly, when calculating the positions of other backbone points above s_1, in order to achieve the effect that the blade of grass is tangent to the boundary sphere when the grass is pushed aside, the direction of the remnant segments above s_1 should point along the direction of vector from s_0 to s_1.



(a) (b)

Figure 21: Sphere-plant interaction. (a) 2D viewing. Red dot represents the original position of backbone point in the blade of grass with no interaction. Green dot represents the center of the boundary sphere. Blue dot represents the position of backbone point with the sphere-plant interaction and with constraint of shape. Yellow dot represents position of backbone point with the sphere-plant interaction and with no constraint of shape. (b) 3D viewing. The grass blades are pushed outside by the gray boundary sphere.

In this method, since we do not use the interaction between the real vertex of model and the real segment points of plant, the accuracy of the interaction is not perfect. However, this simple technique can achieve fast and reasonable result for the sphere and plant interaction. It is sufficient for our purpose since we can obtain both real-time and convincing animation. Figure 21.b shows the snapshot of sphere and plant interaction in a 3D scene.

7 Results

This chapter contains several examples of rendering the large-scale prairie. The project runs in the platform with the following specifications:

MODEL	DELL PRECISION M70
CPU	Intel(R) Centrino II 2.13GHz
Memory	2GB
GPU	NVIDIA Quadro FX Go 1400 with 256 MB dedicated memory
Operation System	Windows XP with SP2
Development Environment	Microsoft Visual C++ with Studio 2003.Net
Graphics API	OpenGL 1.1

Table 3: The specifications for the platform.

Figure 22 shows an aesthetic and convincing scene in our project. The prairie has 16x16 grass grids, and each grass grid contains 20x20 blades of grasses. Each blade of grass has 4 backbone points. The camera position follows the monster model when it walks in the prairie. The blades of grasses which surround the monster are pushed outside by the sphere-plant interaction. All blades of grasses and billboards of grasses are animating with respect to the presence of wind, and the velocity of wind is calculated by the Navier-Stokes equations with 10^3 grid cells. For the close-up of single blade of grass, each blade of grass animates with unique direction and strength. The technique of view frustum culling guarantees that the invisible grasses are not rendered, including the invisible grass grid and invisible single blades of grasses in the visible grass grid. When the camera is traversing from one grass grid to other, the visual effect of fade-in and fade-out provides a smooth transform between billboards and single plants.

Figure 22: The aesthetic large-scale prairie with physically based animation.

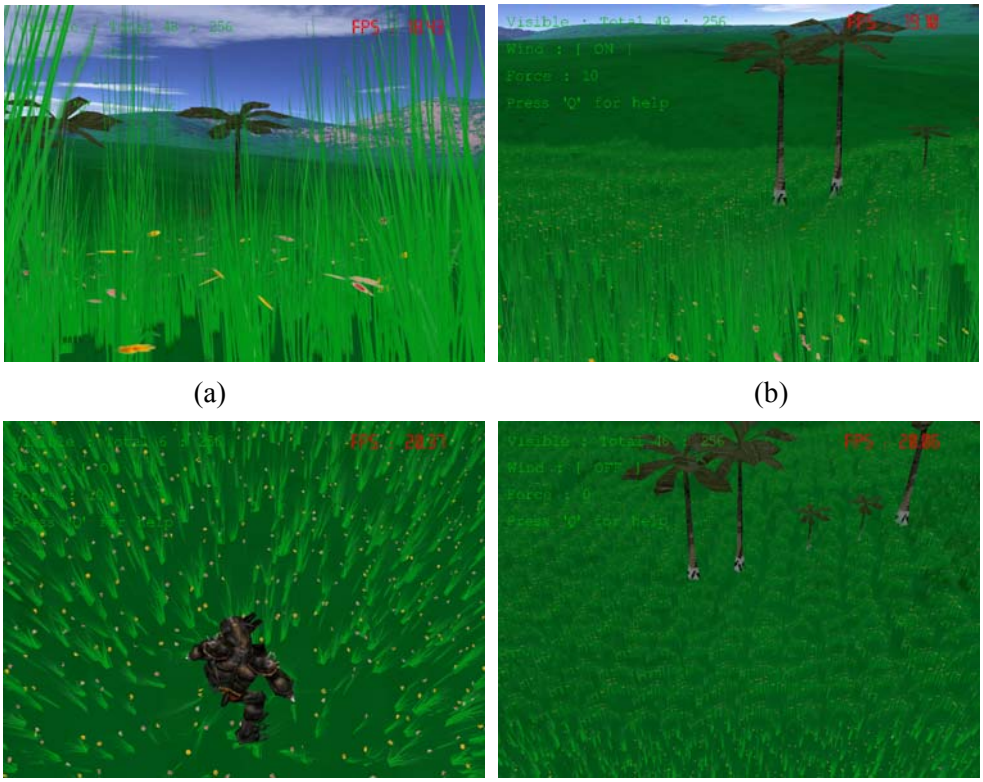
Our design is more advantageous and flexible than any existent applications. For example, in [3], Shinya et al. propose stochastic motion for trees and grasses by using the power spectrum of wind in uniform fields. However, the solution of their application is too time-consuming, thus it is impossible to apply to a large-scale scene such as prairie in our project. Compared with [3], we employ the fast and stable physically-based equations, the Navier-Stokes equations, to describe the realistic motion in the presence of wind and provide real-time animation in the large-scale prairie. In [1], Pelzer et al. propose the meadow with animating grasses. However, the motions of grasses are not realistic since they only apply simple trigonometric functions, such as sine and cosine, to control animating grasses. Moreover, all grasses are represented by billboards so that the shapes are un-natural when the camera moves closer to grasses. Compared with [1], we apply single plants for close-up and billboards for far-distance, and a smooth visual transform is employed if changing the models between single plants and billboards. In [2], Bakay et al. propose the displacement map to simulate the animating grass by rendering several layers overlapped each other and animate

each layer with different direction. However, if the camera moves closer, it is un-natural to see a set of circles which represent the models of grasses. Compared with [2], we use rectangular polygons and complex textures with alpha channel to represent the translucent components of blades of grass, so that this design is more aesthetic and realistic than the method in [2]. In [5], Perbet et al. propose a method for the large-scale prairies in real time. They pre-define several postures of grasses and choose one of them for animation, and place a 2D mask map representing the effect of wind on the prairie from the top view. Compared with [5], we calculate the velocity using physically-based equations for each control points in grasses, thus different control points obtain different velocities. Therefore it is more realistic than simply laying a 2D mask on the prairie in [5].

Furthermore, our design is not restricted to apply for the particular purpose of application that the model is walking through a large-scale prairie, such as the games for outdoor shooting. We have considered different kinds of potential applications associated with the scene of the large-scale prairie and provided an integrated and all-purpose system for them. By easily modifying the height of camera position, we can emphasize different parts of scene which can be adapted to different applications. For example, the lower height of the camera in the prairie can closely look up at the single blade of grass, thus the scene with “huge” animating single blades of grasses and flowers surrounding the camera can be used for simulating the view from an ant’s eye, such as the scene in the movie “A bug’s life”. The higher location of the camera can view the large parts of scene

with animating billboards, which can be used for the aircraft flying over the prairie. In a word, our design of elaborate model of single blades of grasses near camera position and coarse model of billboards distant away from camera position can be used to solve different potential applications concerning the aesthetically animating grasses, while the applications in [1], [2], [3], [5] only pay attention to the particular purposes of animating grasses.

We snapshot four scenes from our project in Figure 23 when the camera moves from the lower position to the higher position.



(a) (b)

(c) (d)

Figure 23: Final rendering results for different views when the height of camera increases. (a) height=1. (b) height=5. (c) height=10. (d) height=20.

8 Conclusion

In this project, we present an integrated model for grass dynamics in a large-scale prairie. The dynamics of grass is controlled by the Navier-Stokes equations which provide simulation of natural wind. We define grasses closer to camera position with single blades, and grasses distant away from camera position with billboards. For both billboards and single blades, we provide correct display from far to near sorting. When the camera looks around the scene, we use view frustum culling to only render the visible grass grids in order to save the computational time. When the camera moves from one grass grid to another, the transformation between billboards and single blades can be shown smoothly. The boundary sphere of model provides the model-plant interaction fast and easily.

The design of the integrated system is easy to tune for different platform based on the multi-platform graphics API. It is also easy to adjust many parameters according to the capacity of CPU and specification of application, such as the segment numbers in a single blade, the numbers of blades in one grass grid, and the numbers of grass grids in whole scene.

In conclusion, our design for the large-scale prairie can achieve the objective of both aesthetic animation and real-time rendering, so this project proves itself useful for the gaming industry.

Bibliography

[1] Kurt Pelzer. "Rendering Countless Blades of Waving Grass". *GPU Gems I*, pages 107-121, 2004.

[2] Brook Bakay, Paul Lalonde, and Wolfgang Heidrich. "Real Time Animated Grass". In *EUROGRAPHICS*, Short Presentations, 2002.

[3] Mikio Shinya and Alain Fournier. "Stochastic motion – motion under the influence of wind". In *Computer Graphics Forum*, 1992.

[4] Jakub Wejchert and David Haumann. "Animation aerodynamics". In Computer Graphics, 1991.

[5] Frank Perbet and Marie-Paule Cani. "Animating Prairies in Real-Time". In *Proceedings of the symposium on Interactive 3D graphics*, 2001.

[6] Nick Foster and Dimitris Metaxas. "Modeling the Motion of a Hot, Turbulent Gas". In *Computer Graphics Proceedings*, Annual Conference Series, 1997.

[7] Jos Stam. "Stable Fluids". In *SIGGRAPH 99*, 1999.

[8] Jos Stam. "Real-Time Fluid Dynamics for Games". In *Proceedings of the Game Developer Conference*, 2003.

[9] Richard Barrett, Michael Berry, Tony F. Chan, James Demmel, June M. Donato, Jack Dongarra, Victor Eijkhout, Roldan Pozo, Charles Romine, and Henk Van der Vorst. "Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd edition". Philadelphia, PA: SIAM, 1994.

[10] Euler Integration. http://en.wikipedia.org/wiki/Euler_integration.

[11] Cantarella, Jason, DeTurck, Dennis, Gluck, Herman. "Vector calculus and the topology of domains in 3-space". American Mathematical Monthly, 2002.

[12] Daniel Zwillinger. "Handbook of Differential Equations, 3rd edition". Boston, MA: Academic Press, page 129, 1997.

[13] Richard H. Bartels, John C. Beatty, and Brian A. Barsky. "An Introduction to Splines for Use in Computer Graphics and Geometric Modeling". Chapter. 10. San Francisco, CA: Morgan Kaufmann, pages 211-245, 1998.

[14] Verlet Loup. "Computer experiments on classical fluids. I. Thermodynamical properties of Lennard-Jones molecules". Phys. Rev. 159, pages 98-103, 1967.

[15] Thomas Jakobsen. "Advanced Character-Physics". Proceedings of the Game Developer Conference, 2001.

[16] "OpenGL Programming Guide, 2nd edition". Addison-Wesley Publishing Company, pages 162-172, 1997.

[17] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. "Introduction to Algorithms, 2nd edition". MIT Press, 2001.

[18] Radu Privantu. Height Map Editor, http://hme.sourceforge.net/.

[19] Bilinear interpolation. http://en.wikipedia.org/wiki/Bilinear_interpolation.