CG Final Report

CHENG Yifeng, HU Xinyue, SONG Haoyang

April 2022

Contents

1 Particles

When playing the game, there will be a lot of particle effects appear, such as the hitting tree effect, smoking from the smoke pipe of the running car and energy absorption effect. The particle effect is created in a memory-friendly and highly malleable way by using two arrays and the TweenMax.min.js package.

TweenMax package gives us a good method to create a gradient animation by using a source position and a destination position. It can also define the feature of gradient animation like the effect of ease and speed of animation. The single-particle explodes from the center and moves to its destination followed by the changing of scale and rotation along the way. Moreover, after the particle finishes moving, the package also provides an "on complete" method to modify the particle after completing. This "on complete" method is used in our program to remove the mesh of the particle from the scene and push the particle into the particle pool which will be discussed later.

Two essential arrays named InUseArray and PoolArray are used to store the particles. InUseArray contains all using particles including the removing particles and PoolArray contains the particles which are not in the scene and ready to be used again. The recycle particle technique frees the memory press because there will be only a few reused particles utilized in the game and there is no need to create extra particles to occupy the memory of the computer. The procedure can be described as a particle is needed when there is a particle effect needed and the particle is either gotten by newing a particle or popping from pool. After the particle finishes the processing, it will be pushed back to the pool for further processing.

2 Game Design And Interaction

Many important game parameters are stored in a map for reference. The parameters include life, game speed, enemy hitting tolerance distance, energy lost speed, etc. The game can be easily modified by changing the number of the parameters and creating the game with a desirable style.

Such as the life parameter, taking the running car as an example, the parameter is first used to initialize the life of the running car. By referring to the life of the car, the smoke effect could generate different colors, scales, and densities of the particle. As well as the speed parameter, if the speed is high, the overall pace of the game will be on the fast side and vice versa.

3 Hierarchical Modeling

The building of different kinds of objects using Hierarchical modeling to make it easy to copy and modify. Our basic principle is using a THREE.Obejct3D to include other THREE.Object3D or other objects and by modifying the root object3D, the effect is also applied to the children.

Take the creation of trees as an example. A ThreeConesTree is an object3D and it contains 3 cones and 1 box. A BoxTree is an object3D and it contains 2 boxes. All these kinds of trees are added to another object3D named ground. The ground will rotate in the first game and the children of it also will rotate with it at the same speed.

Similar to the tree, the airplane is also being built using hierarchical modeling, just more sub components.

4 Camera Control

In the phase 1 of our game, our camera is fixed. In the phase 2 of our game, the camera is moving with the player. The position of the camera is relative to the position of the airplane. Our airplane is being controlled by keyboard "WASD" for rotation. In each animation loop, the position of the air plane is being updated with a little translation determined by the player's keyboard input. As the camera is following the airplane, the translation of airplane should also being applied to the camera. If taking the airplane's model coordinate system as a reference, a vector (refer as model-coordinate camera translation) should be applied to the camera to translate from the origin(airplane). Apply the airplane's model matrix to "model-coordinate camera translation", is camera's translation in the world, makes camera always move with the airplane.

5 Terrain Generation System

The most important technical novelty is our terrain generation system. We neither explicitly modeled the landscape nor import models. The game world is infinitely large, when the player moves around, the terrain is generated and loaded dynamically. To implement the infinite world, we segment the world into different partitions, which we called chunk.

5.1 Perlin Noise

We user the noise generation method invented by Ken Perlin. In our game, we are using 2d Perlin noise. The calculation of Perlin noise is intuitive. On the 2d plane, on the integer grid randomly generate a gradient vector, notice that these gradient vectors are not completely random, but are composed of 12 vectors from the center point of the unit cube (3d) to the midpoints of each edge. Then, for each point inside the integer grid (a unit square), there are four distance vectors (from integer grid point to the point in the unit square). Then we can get four dot products of distance vector and random gradient vector. The final noise calculated at the point is interpolation of four dot products. There are many functions to use, we are using the "Fade Function" to interpolate four product values.

$$6t^5 - 15t^4 + 10t^3 \tag{1}$$

Notice that on the integer grid, the Perlin noise calculated is always zero, when sampling in the Perlin noise space for vertices, to avoid always sampling from the integer grid position, we add a small offset.

5.2 Chunk

For each chunk, it holds the geometries inside of a cubic space. The x-z plane is being segmented into different chunks. We have define several parameters to describe properties of a chunk.

- chunkSize: Controls the size of the chunk, it defines the width of the square space it takes in the x-z plane
- chunkSegments: Used to control the density of vertices to construct the terrain geometry, the higher, the finer looking of the terrain, but consumes more computational power.
- landHeight: Controls the amplitude of the chunk.
- treeGenerate: The probability of tree generation.
- seed: Seed for Perlin noise, which decides the landscape of our world
- samplingScale: Controls the scale for sampling in the Perlin noise space. The larger the sampling scale, the larger space sampled inside the Perlin noise space for a single chunk. Also affects the looking of the chunk.
- loadDistance: Can be understand as the visibility distance. Defines the distance from the player a chunk will be generated and loaded to the world.
- destructionDistance: Defines how far a chunk away from the user will be be destroyed

5.2.1 Chunk Coordinate

As each chunk, it takes a squre space in the x-z plane, so we give each chunk a separate coordinate, the chunk coordinate. The chunk coordinate is describes the south west corner a chunk. (take x direction as north). The relation between the chunk coordinate and world coordinate is:

$$(x_{chunk}, z_{chunk}) = (\frac{x_{world}}{\text{chunkSize}}, \frac{z_{world}}{\text{chunkSize}}) \tag{2}$$

The x_{world} and z_{world} refer the south west corner of the chunk. Using the chunk coordinate, make the chunk easier to manage in later dynamic chunk loading. Also for the object inside a chunk, we are using the model space coordinate, by adding a translation derived from chunk coordinate and chunkSize, its can be changed to world coordinate easily.

5.2.2 Terrain Geometry Generation

The Perlin noise is used for crafting the terrain geometry. Initially, the chunk only have a THREE.PlaneBufferGeometry. We need to manipulate the y-position (vertical) of vertices to change the shape of the plane, make the plane have a mountain like looking. Here, we are using the 2d Perlin noise. The 2d Perlin noise takes two parameters, which are the x and z coordinates. It will returns a noise value for point we taken in. For each vertex in our plane, we take its world coordinate to the Perlin noise space, get the noise value then change its y-coordinate.

5.2.3 Terrain Decoration

Tree Generation After the terrain is generated, we need to decorate the terrain. First of all, is the tree decoration. Using the THREE.PlaneBufferGeometry, we can get the the model coordinate of each vertex, also vertex normal. With such information, we can plant the tree to our mountain. Also, for those vertices that is lower than the water surface, we simply skip them. Whether the there will be a tree at a specific vertex or not is being decided by the vertex height (it should above the water surface) and the Perlin noise score (if the Perlin noise at the position is higher than a specific value, a tree will be generated). There are some randomness for decision of type of the tree. We have four types of tree, and we define four types of tree distribution. Each distribution having one type of tree as majority. The Perlin noise score will decide which distribution we use. And we wil randomly draw a tree from that distribution.

Cloud Generation Similar to the tree generation, if the Periln noise score at the position of the vertex is higher than a specific threshold, a cloud object will be generated. The construction of cloud object itself is on a random basis (in terms of the rotation of the cube, number of cubes).

5.2.4 Water

The wave of water is realized by the Gerstner waves function. Following will start with the original formula and then present a simplified version which will be easier to implement in code. In order to make the game closer to real-time, the calculations are done in the vertex shader, where the game time is passed in as a parameter.

The free surface is described parametrically as a function of the parameters α and β, as well as of time t. Specifically: $x = \xi(\alpha, \beta, t), H = y = \zeta(\alpha, \beta, t)$ and

$z = \eta(\alpha, \beta, t)$ where

$$\xi = \alpha - \sum_{m=1}^{M} \frac{k_{x,m}}{k_m} \frac{a_m}{\tanh(k_m h)} \sin(\theta_m), \eta = \beta - \sum_{m=1}^{M} \frac{k_{z,m}}{k_m} \frac{a_m}{\tanh(k_m h)} \sin(\theta_m),$$
$$\zeta = \sum_{m=1}^{M} a_m \cos(\theta_m), \text{and } \theta_m = k_{x,m}\alpha + k_{z,m}\beta - \omega_m t - \phi_m \tag{3}$$

Note that *tanh* is the hyperbolic tangent function, M is the number of wave components considered, a_m is the amplitude of component $m = 1...M$ and ϕ_m is its phase. Also, $k_m = \sqrt{(k_{x,m}^2 + k_{z,m}^2)}$ is its wavenumber and ω_m is its angular frequency.

Furthermore, k_m and ω_m have to satisfy the following constrain: $\omega_m^2 = g k_m \tanh(k_m h)$, with h the mean water depth. In deep water ($h \to \infty$) the hyperbolic tangent goes to one. Horizontal wavenumber vector $\boldsymbol{k}_m$ determine the wave propagation direction of component m. The choice of the various parameters a_m, $k_{z,m}$, $k_{x,m}$, ϕ_m and a certain mean depth h determines the form of the ocean surface. The normal vector $\boldsymbol{n}$ to the surface can be computed using the cross product as:

$$\boldsymbol{n} = \frac{\partial \boldsymbol{s}}{\partial \alpha} \times \frac{\partial \boldsymbol{s}}{\partial \beta} \quad \text{with} \quad \boldsymbol{s}(\alpha, \beta, t) = \begin{pmatrix} \xi(\alpha, \beta, t) \\ \zeta(\alpha, \beta, t) \\ \eta(\alpha, \beta, t) \end{pmatrix} \tag{4}$$

Normal vectors can be useful if we select the Blinn-Phong reflection model in the fragment shader, but we decided to simplify the color assignment by mixing blue and white, with color weights depending on height. Another simplification is that, to avoid the complexity of the above formula in our mini-game, we chose to approximate it using the following version:

$$\mathbf{P}(x, y, t) = \begin{pmatrix} x + \sum (Q_i A_i \times \mathbf{D}_i \cdot x \times \cos(w_i \mathrm{D}_i \cdot (x, y) + \varphi_i t)), \\ y + \sum (Q_i A_i \times \mathbf{D}_i \cdot y \times \cos(w_i \mathrm{D}_i \cdot (x, y) + \varphi_i t)), \\ \sum (A_i \sin(w_i \mathbf{D}_i \cdot (x, y) + \varphi_i t)) \end{pmatrix} \tag{5}$$

In this equation, x, y is actually corresponding to x, z coordinate. A_i is the amplitude, Q_i is the steepness parameter of the wave, $\mathbf{D}_i$ is the wave direction. w_i is the parameters that control the wavelength and ϕ_m is its phase. Before implementing the Gerstner waves function, we also tried to simply use the sine waves function to simulate the water wave.

$$\mathbf{H}(x, y, t) = \sum (A_i \times \sin(\mathrm{D}_i \cdot (x, y) \times w_i + t \times \varphi_i)) \tag{6}$$

But Gerstner's effect is more realistic than sinusoidal functions, because in Gerstner the peaks are steep and the troughs are flat (real world water fluctuations!), while troughs and peaks are symmetric in the sine function.

5.2.5 Destruction

A chunk actually is a complex 3d mesh wit hierarchical modeling. The chunk it self is an THREE.Obeject3D at the root of the scene graph. If a chunk is going to be destroyed, all of its child components should be destroyed too. THREE.js provides function called "traverse" to traverse the scene graph. We use this function to delete all sub component in the scene graph recursively. For each chunk object, it has a "destroy" function so that it can be deleted by a simple function call.

5.3 Dynamic Chunk Loading

For each chunk, it has a chunk coordinator as demonstrated previously. If the chunk coordinate is determined, the shape of the chunk is also determined, as the size and and chunk coordinate will determine the world coordinate, with same world coordinate, the parameters takes in Perlin noise is the same, thus generate terrain generated is the same. Here, we are using a specific class called chunk loader to generate/destroy chunk with moving of the player. The coordinate of the player is just the coordinate of the camera. In the animation loop, there is a funtion called "generateChunk" inside the chunkLoader will be called. "generateChunk" function is responsible for updating the chunk(e.g. time for water calculation), and monitoring the coordinate of the player. The coordinate of player will be parsed to chunk coordinate as shown in Equation2. If the chunk coordinate of the player is changed, meaning the player moves to a new chunk, the chunk loader will check the surrounding chunk aroud the player. The checking area is a square with size of $(2 \cdot \text{loadDistance} + 1) \cdot \text{chunkSize}$ with player in the center chunk of the squre area. If any of the chunk inside this area is void, the chunk loader will generate a chunk and fill the void area. Since our game having limited resource, we can not keep generating chunks, otherwise we will running out of memory and the program will down. If the chunk is too far away from the player, the chunk will be destroyed by the chunkLoader by calling the "destroy" function in 5.2.5 of the chunk.

6 Weather and Season

6.1 Weather particles in the sky

The particles shapes are solely based on Threejs Primitives and they are super simple. In Game1, the particles in the sky (snow, rain, etc.) use the Tetrahedron Geometry. In Game2, the flowers in Spring use the Extrude Geometry; the rain in Summer uses a mix of the Cone Geometry and the Sphere Geometry; the balloons in Autumn use a mix of the scaled Sphere Geometry and a Cylinder Geometry; the snow in Winter also uses the Tetrahedron Geometry. Those particle are randomly generated and rotates as they fall. When particles fall very low, they regenerate (move to high ground) and then fall again, so that

there are always a certain number of particles in space. Finally, the energy balls in the second game use Icosahedron Geometry (20 sides) with golden color.

6.2 Season switching

Season switch makes use of the time in the rendering loop, using the indicator (=sin(time/scale)), if the current indicator has a different sign with the value cached in previous round, season change function will be invoked.

7 Music

7.1 Background music and Season music

Background music will be played when player enters the game. Season music will be played when the season changes, four versions correspond to Spring, Summer, Autumn and Winter. Music are all excerpted from *Stardew Valley*.

7.2 Collision sound effects

Sound will be played when there is a collision detected. In Game 1, when the car hits the tree, the energy of player will decrease so the sound effect is about glass broken; In Game 2, when the helicopter hits the energy ball, the energy will increase so the sound effect is about energy absorption.

8 Landscape color gradient (excluded)

Landscapes with color gradients can be more beautiful, our team tried to change the colors of the terrain itself by mixing some colors based on the (x,z) coordinates of the vertices, and have tried several basic functions to determine the weights of different colors. The effect turns out to be not so good, maybe a more complex formula is needed in determining the color weight in order to make it beautiful. But hemispheres light already brings a little bit of a color gradient, although the color gradient is through the lighting effect rather than the terrain itself.

9 Work Distribution

Everyone in our group is working hard. We have a even work distribution. We are all contributing to the whole project.

CHENG Yifeng (33.3%): Tree generation(phase 1). Process Generation of Landscape, Chunk and Chunk Loader. Airplane Modeling. Camera control (phase 2).

HU Xinyue (33.3%): Water, Weather, Season, Music, and Color Gradient.

SONG Haoyang (33.3%): Particles, Game Design and Interaction, Hierarchical Modeling