MSc CGE – Maths&Gfx2 – Coursework3:

Interactive Terrain Generation and Effective Water Simulation

Team Members: Wanganning Wu, Alex Lux

Number of pages: 15

Date of Submission: 12/04/2014

Introduction

The goal of this group project is to create smooth terrain surface which is editable and generated interactively, and to effectively simulate water surface.

There exists many algorithms to generate terrain. The fractal landscape is a common idea and choice, which is a surface generated using a stochastic algorithm designed to produce fractal behaviour that mimics the appearance of natural terrain. One of the frequently-used stochastic algorithms for the fractal landscape generation is the Diamond-square algorithm. It is a good choice to procedurally generate the terrain, but it is less controllable due to its stochasticity, and the terrain generated by this algorithm has noticeable vertical and horizontal "creases" due to the most significant perturbation taking place in a rectangular grid.

As NURBS is derivable in its knot span, the surface constructed by NURBS is smooth at any point on the surface. While the control points and weights of NURBS are two properties that affect the shape of the surface, they can be manipulated through the user interface intuitively at the same time. Having these features, the NURBS surface is chosen and implemented to generate the terrain in this project.

To effectively simulate realistic looking water, there are techniques such as the Fast Fourier Transform, Gerstner Wave and summation of different sine waves. There are benefits in each of the domains. For example FFT as the title suggest it is very fast in computation for complex-number arithmetic to group theory and number theory, there are many application uses FFT such as data compression, signal processing, wave simulation and many more.

In specific to the domain of wave simulation the advantage of FFT is the flexibility to mimic and recreate any type of wave characteristic and its fast computational speed for calculating transforms. As for summation of sine waves it is very simple and straight forward to implement because the geometric undulation of the surface mesh is easily represented by the sum of simple periodic waves. Lastly for Gerstner wave it is extension of summation of sine wave approach but offer a particular characteristic. This characteristic is shown when forming sharper crest, as the formula compute the X and Z axis of the vertices, the vertices move towards each crest so wave crests that are the sharpest have the highest frequency of polygon count.

For this particular application to simulate ocean waves, we will be focusing on implementing the summation of sine waves and Gerstner Wave approach. The reasoning behind in choosing these two techniques would be that they show close relationship in terms of the mathematical formula and there maybe more self similarities and differences to be explored during implementation.

Theory: NURBS

Non-uniform rational basis spline (NURBS) is a mathematical model commonly used in computer graphics for generating and representing curves and surfaces. It offers great flexibility and precision for handling both analytic (surfaces defined by common mathematical formulae) and modelled shapes.

As NURBS is a generalization of B-spline, which in turn is generalization of Bezier curve, the definition of the B-spline is described below first:

Given $n + 1$ control points $\mathbf{P}_0$, $\mathbf{P}_1$, ..., $\mathbf{P}_n$ and a knot vector $U = \{ u_0, u_1, ..., u_m \}$, the B-spline curve of degree p defined by these control points and knot vector U is

$$\mathbf{C}(u) = \sum_{i=0}^{n} N_{i,p}(u)\mathbf{P}_i$$

Where Ni,p is the the B-spline basis function, and C(u) is the point on the curve with respect to the corresponding u parameter.

Through applying a series of discrete u value to this formula, we can get a curve formed by these discrete sampling points.

The B-spline basis function is defined as below.

$$\begin{aligned} N_{i,0}(u) &= \begin{cases} 1 & \text{if } u_i \le u < u_{i+1} \\ \\ 0 & \text{otherwise} \end{cases} \\ N_{i,p}(u) &= \frac{u - u_i}{u_{i+p} - u_i} N_{i,p-1}(u) + \frac{u_{i+p+1} - u}{u_{i+p+1} - u_{i+1}} N_{i+1,p-1}(u) \end{aligned}$$

From the above definition we can see, as the basis functions of B-spline are defined recursively, it can be easily implemented with recursion in code. But for the sake of the space expenses and time efficiency, the generation of the basis function is implemented with loop statements iteratively in this class.

After applying weight value to the non-rational B-spline definition, we can obtain the NURBS definition as below.

$$\mathbf{C}(u) = \frac{1}{\sum_{i=0}^{n} N_{i,p}(u) w_i} \sum_{i=0}^{n} N_{i,p}(u) w_i \mathbf{P}_i$$

With the definition of B-spline curve, we can then extend to the definition of B-spline surface. Below is the formula of B-spling formula.

$$\mathbf{p}(u,v) = \sum_{i=0}^{m}\sum_{j=0}^{n} N_{i,p}(u)N_{j,q}(v)\mathbf{p}_{i,j}$$

where $N_{i,p}(u)$ and $N_{j,q}(v)$ are B-spline basis functions of degree p and q, in two directions of the surface respectively.

From the definition of the B-spline surface, we can see that the inner part of the formula is a B-spline curve formula in one direction of the surface, which provide control points for outer part of the formula which is also a B-spline curve formula in the other direction of the surface. In order to obtain a point from the surface, we should at least provide two parameters which are u and v in the formula to locate the point in the surface.

Similarly, we can apply weight value to the B-spline surface definition to get the definition of the NURBS surface as below.

$$S(u,v) = \sum_{i=1}^{k}\sum_{j=1}^{l} R_{i,j}(u,v)\mathbf{P}_{i,j}$$

where $$R_{i,j}(u,v) = \frac{N_{i,n}(u)N_{j,m}(v)w_{i,j}}{\sum_{p=1}^{k}\sum_{q=1}^{l} N_{p,n}(u)N_{q,m}(v)w_{p,q}}$$

Theory: Effective Water Simulation:

For the water simulation, I have started off with a single sine wave and gradually implemented a sum of 4 different sine waves to vary the heights of the wave. To further manipulate the sine wave, there are specific parameters to calculate. These are:

1. **WaveLength (L):** This is the distance between each wave in world space. The frequency formula for a single wave is w **= 2π/**L.
2. **Amplitude (A):** The height from the water plane to the wave crest.
3. **Speed (S):** The distance the crest moves forward per second. This is expressed as "phase-constant". The formula for the speed is Phase-Constant = S x 2π /L
4. **Direction (D):** The horizontal vector perpendicular to the wave.

For each wave, the function of horizontal position (X, Y) and time (t) is defined as the following equation:

$$W_i(x,y,t) = A_i \times \sin\left(\mathbf{D}_i \cdot (x,y) \times w_i + t \times \varphi_i\right).$$

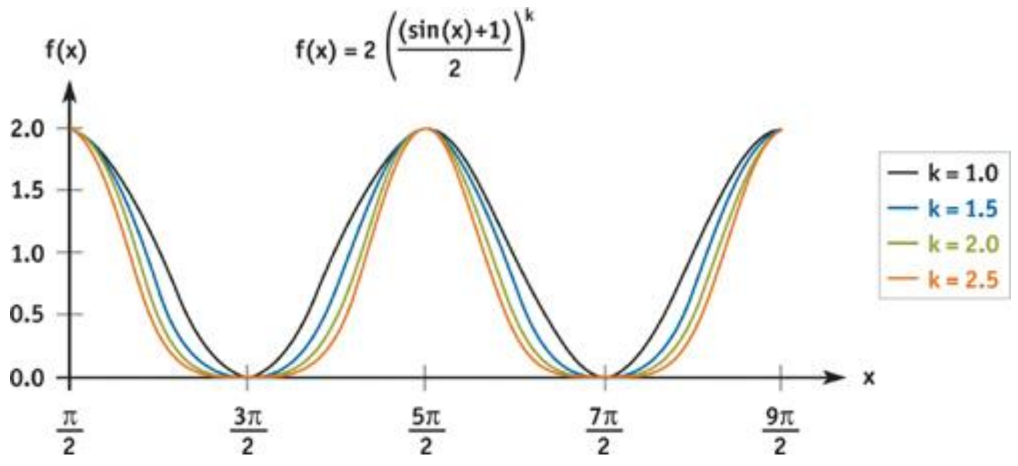
For the total surface, it is the sum of the above equation:

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

The characteristic of using the sum of sine waves in this situation would produce very smooth round waves. In order to simulate other waves that have sharper peaks and wider troughs, it is possible by offsetting the above sine function to be nonnegative and increase the sine value to an exponent “k” which defines the steepness of the wave.

$$W_i(x, y, t) = 2A_i \times \left(\frac{\sin\left(\mathbf{D}_i \cdot (x, y) \times w_i + t \times \varphi_i\right) + 1}{2} \right)^k,$$

To visually give a better understand of the exponent effect, below is a graph to represent the change of the wave in respect to the exponent value.



In this ocean simulator, I have only combined 4 different waves but there is no limit in the number of waves. The increase of different property waves gives more variety and detail but at the cost of more calculations. A good optimization would be sending the vertex calculation for the vertex shader to handle.

Directional and Circular Waves

Directional waves are often preferable in large bodies of water such as ocean or sea because they simulate a better model for wind driven waves. As for circular waves, it is better suited for smaller bodies of water such as pond or swimming pool because circular waves have the unique characteristic of non repeated interference patterns.

Gerstner Wave Formula

Gerstner wave formula is an extension of combining sine waves together to form an effective wave. Not only does it calculate the Y axis but it also calculate the x and z axis to move vertices toward each crest to form sharper crest.

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

Team Work

We split the work into two main tasks which are the terrain generation for Wanganning and the water simulation for Alex. The two tasks were then divided into smaller tasks which are implemented by each of the team members at a time. During the development, we held meetings and discussed the problems and progress so far and helped each other. Below are the lists of tasks for each team member:

Tasks Assigned: Wanganning Wu

- Implementing a NURBS surface class that supports the generation of the NURBS surface terrain.
- Implementing the generation and rendering of the NURBS surface terrain.
- Implementing a feature to drag and move a control point along a plane which is parallel to the camera projection plane at any viewing angle and point in the scene with mouse and change its weight with keyboard.
- Implementing a ray detection feature to select between control points at any viewing angle in order to change the corresponding weight and position;
- Implementing a skybox.
- Implementing a camera control class to enable viewing the terrain and control points with mouse or keyboard.

Tasks Assigned: Alex Luk

- Implementation of the base class “Water_Square” to support the generation and rendering of the water simulation.
- Implementation of the main water simulation class that handles and compute the core components to draw the procedurally generated water plane.

Implementation

Class Overview

This section describes the classes that have been implemented and their main functionalities. Below are the classes implemented:

terrain_app	nurbs_surface	sky_box
camera_control	Water_Square	Water_Mesh_app

Class Introduction

class terrain_app

This is a custom app class derived from class app that maintains shader instance, nurbs_surface instance, camera_control instance, responds keyboard and mouse messages, selects and manipulates control points with the computation of ray casting and matrix transformation, prepares rendering states and renders control points and terrain meshes for the NURBS surface terrain.

class nurbs_surface

This class is the core class that contains the necessary components for the NURBS formula. It provides functions that initialize the component variables of the NURBS formula(such as the basis functions, knot vector, weights, control points, and degree), compute the NURBS formulae to get the basis functions and obtain a specific vertex on the surface according to the u and v parameters, which can then be used to form the terrain mesh.

class sky_box

This class mainly maintains the skybox of the scene. The relevant work it does includes the initialization of the skybox, loading skybox texture with a path string, creating vertices to form the skybox mesh, assigning uv coordinates to each vertex, setting the position of the skybox, and rendering the skybox.

class camera_control

This is an auxiliary class that maintains camera_to_world matrix. It offers member functions to change the distance between the camera position and the object being viewed, and rotate the camera around the object. All the changes will be applied to the matrix straight away rather than construct the matrix each frame as the frequency of the change of camera parameters is relatively low compared to the frame rate.

class Water_Square

This class handles the drawing of a single quad mesh and a group of getters and setters method to return the position of the four quad vertices in word space.

class Water_Mesh_app

This class handles the procedural generation of the water mesh plane. It includes several helper functions to compute the mathematical formula for each of the core elements that affects the water simulation.

Implementation details

This section describes the member functions of classes in in details.

class terrain_app

terrain_app()

This is the constructor of the class which does basic initialization for this class. The work includes the initialization of member variables such as last mouse coordinates, keyboard cool down time, boolean value that toggles the visibility of the wireframe mode and the visibility of the control points, and the index of the current selected control point.

void app_init()

This is an member function which will be invoked after OpenGL is initialized. It does initialization work such as setting the rendering state, loading skybox and terrain textures, adding knots to both knot vectors to be used in NURBS formula, invoking the function that creates control points for the NURBS surface, initializing shaders and transformation matrix.

void create_ctrl_points()

This function creates control points for the NURBS surface, adds weight values which are related to basis functions, and sets the initial color for each control point.

void view_to_world(const vec3 *v)

This function takes in the view space coordinate of a point and returns its corresponding coordinates in the world space. It is mainly used in the selection and manipulation of the control point.

void select_ctrl_point()

This function does the relevant work that helps to pick the control point the mouse cursor is pointing to when left mouse button is pressed. The basic idea is to obtain the ray starting from the camera position and pointing to where the mouse cursor is pointing to, and check intersections of the ray and the bounding sphere of each control point. If there is at least one intersection with a bounding sphere of a control point, then the corresponding control point is the one to be selected, and it will highlight the corresponding control point and reset the previous selected control point's color. The ray detection process is listed below.

- According to the projection frustum parameter, compute the corresponding view space coordinates of the screen coordinates where the mouse cursor is pointing to. The view space coordinates can be any coordinates as long as it is on the ray casted from the camera.
- Transforming the view space coordinates back to the world space coordinates using the inverse of the view matrix, and normalize the vector starting from the camera position **v0**(x0, y0, z0) and ending at this transformed world space coordinates to get the unit vector **v1**(x1, y1, z1) of the ray in the world space. So the ray can be represented as **v0** + k * **v1 =** (x0 + kx1, y0 + ky1, z0 + kz1)
- Specify a constant variable R which is to represent the radius of the bounding sphere of a control point. As the sphere equation is (x – a)^2 + (y – b)^2 + (z – c)^2 = R^2, where a, b, c are the corresponding coordinates of the centre of sphere on X-axis, Y-axis and Z-axis. After we replace the unknowns with the formula of the ray above, we can get a quadratic equation with one unknown: (x0 + kx1 – a)^2 + (y0 + ky1 – b)^2 + (z0 + kz1 – c)^2 = R^2.
- The root count of this equation is equal to the intersection count of the ray and the bounding sphere, so we can use the discriminant b^2 – 4ac to get how many roots or intersections there are. If b^2 – 4ac >=0, there is at least one intersection between the ray and the bounding sphere, and the corresponding control point is the one that needs to be selected.

```
void select_ctrl_point()

{
    int x, y, w, h;
    get_mouse_pos(x, y);
    get_viewport_size(w, h);
    vec3 target(x - w * .5f, h * .5f - y, -w * .5f);
    vec3 camera((vec3&)(cc.get_matrix()[3]));
    vec3 ray((view_to_world(target) - camera).normalize());

    static float radius = .05f;

    const dynarray<vec3> &points = terrain.get_ctrl_points();
    float x0 = camera[0], y0 = camera[1], z0 = camera[2], x1 = ray[0], y1 =
ray[1], z1 = ray[2];
    for(unsigned int i = 0; i < points.size(); i++)
    {
        float x02 = x0 - points[i][0];
        float y02 = y0 - points[i][1];
        float z02 = z0 - points[i][2];

        float a = x1 * x1 + y1 * y1 + z1 * z1;
        float b = 2 * (-x1 * x02 - y1 * y02 - z1 * z02);
        float c = x02 * x02+ y02 * y02 + z02 * z02 - radius * radius;

        if(b * b - 4 * a * c >= 0)
        {
            ctrl_point_colors[current_selected_ctrl_point] = vec3(1, 0, 0);
            current_selected_ctrl_point = i;
            ctrl_point_colors[current_selected_ctrl_point] = vec3(1, 1, 0);
        }
    }
}
```

void draw_ctrl_points()

If the toggle Boolean variable controlling the visibility of the control point is true, this function will be invoked in the main rendering function. It enables the vertex_color_shader to draw the control points with pure color.

void generate_terrain_mesh(int resolution)

According to the resolution argument taken, this function will decide how many sampling points it will sample on the surface and generate the same amount of u, v coordinates for sampling. Invoking the interface provided by the nurbs_surface class with specific u, v coordinates passed in, a point will be sampled from the NURBS surface. All the points sampled will then be connected as a complete terrain mesh and rendered in the quad-rendering mode.

void move_ctrl_point(int x, int y)

This function is implemented to achieve the interactive feature of the terrain generation. It enables moving the control point along the plane which is parallel to the camera projection plane at any viewing angle and point in the scene. It means that through mouse input the user can intuitively drag and move the control point at any viewing angle only if the control point can be seen on the screen. In this way, the user can move the control point to anywhere through rotating view angle and dragging the control point on the screen. The process is explained as below:

- According to the projection frustum parameter and the end point of the cursor track on the screen, compute the corresponding view space coordinates of the screen coordinates of the end point. The view space coordinates can be any coordinates as long as it is on the ray casted from the camera.
- In order to figure out where the ray intersects with the plane the control point will be moved along, the world space coordinates of the control point should be transformed to the view space to get its z coordinate in the view space.
- After the z coordinate of the control point is obtained, as we are moving the control point along the plane that is parallel to the camera projection plane, the corresponding z coordinate of the end point of the cursor track is actually the same as the control point's z coordinate in the view space. So we can put the z coordinate into the ray formula v0(x0, y0, z0) + k * v1(x1, y1, z1) to get the value of coefficient k, and then we can get the x and y coordinate of the intersection point between the ray and the plane we want the control point to move along in view space.
- After the coordinates in the view space are transformed back to the world space, we get the corresponding world position to which the control point is moved.

```
void move_ctrl_point(int x, int y)
{
        if(mouse_x != x || mouse_y != y)
        {
            const dynarray<vec3> &ctrl_points = terrain.get_ctrl_points();
            if(current_selected_ctrl_point < ctrl_points.size())
            {
                int w, h;
                get_viewport_size(w, h);
                float half_w = w * .5f;
                float half_h = h * .5f;
                vec3 ray(x - half_w, half_h - y, -half_w);
                mat4t worldToCamera;
                cc.get_matrix().invertQuick(worldToCamera);
                const vec3 &ctrl_point =
ctrl_points[current_selected_ctrl_point];
                vec3 v =  ctrl_point * worldToCamera;
                float k = v[2] / ray[2];
                v[0] = ray[0] * k;
                v[1] = ray[1] * k;
                v[2] = ray[2] * k;
                terrain.set_ctrl_points(current_selected_ctrl_point,
view_to_world(v));
            }

            mouse_x = x;
            mouse_y = y;
        }
}
```

void draw_world(int x, int y, int w, int h)

This is the main rendering function of this project. It updates the FPS of this project on the caption of the window, invokes function that handles the mouse and keyboard messages, regenerates the terrain mesh according to the user input(the changes of weights and positions of the control points), the builds the transformation matrix, sets the rendering states and parameters, and renders the skybox, control points, and terrain.

void handle_messages()

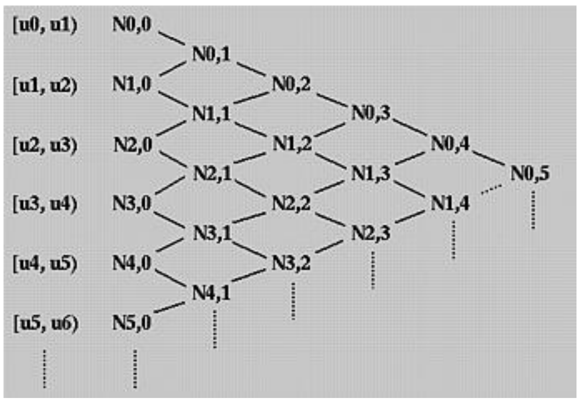
This function is in charge of both mouse and keyboard message handling.

Keyboard	**Function**
S	Decrease the camera view distance
W	Increase the camera view distance
C	Toggle the visibility of the control points
P	Toggle the wireframe rendering mode
key_up	Increase the weight at the selected control point
key_down	Decrease the weight at the selected control point
Mouse	**Function**
Left Button Click	Select a control point
Left Button Drag	Move the selected control point
Right Button Drag	Rotate the camera

class nurbs_surface

void get_basis_functions(int degree, float t, const dynarray<float> &knots, dynarray<float> &basis)

This function is the core function in this class. It takes in four necessary components of a B-spline, which are the degree, the time variable, the knot vector and the control point count, to compute the basis functions of a B-spline. According to the triangular computation scheme(illustrated with the figure below), the computation of the basis function is implemented with loop statements iteratively in this function.



```
void get_basis_functions(int degree, float t, const dynarray<float> &knots,
dynarray<float> &basis)
{
    unsigned int knot_count;
    unsigned int ctrl_point_count = basis.size();
    float d,e;
    knot_count = knots.size();

    for (unsigned int i = 0; i < knot_count - 1; i++){
        if (( t >= knots[i]) && (t < knots[i + 1]))
            temp_basis[i] = 1;
        else
            temp_basis[i] = 0;
    }
    /* calculate the higher order basis functions */
    for (int k = 1; k <= degree; k++){
        for (unsigned int i = 0; i < knot_count - k; i++){
            if (temp_basis[i] != 0)    /* if the lower order basis function
is zero skip the calculation */
                d = ((t-knots[i])*temp_basis[i])/(knots[i + k]-knots[i]);
            else
                d = 0;
            if (temp_basis[i + 1] != 0)     /* if the lower order basis
function is zero skip the calculation */
                e = ((knots[i + k+1]-t)*temp_basis[i + 1])/(knots[i + k +
1]-knots[i + 1]);
            else
                e = 0;
            temp_basis[i] = d + e;
        }
    }
    if (t == knots[knot_count - 1] || t > knots[knot_count - 1] && (t -
knots[knot_count - 1]) < .00001 ){       //    pick up last point
        temp_basis[ctrl_point_count - 1] = 1;
    }
    for(unsigned int i = 0; i < ctrl_point_count; i++) {
        basis[i] = temp_basis[i];
    }
}
```

void get_surface_vertex(vec3 &vertex, float u, float v)

This function invokes the **get_basis_functions** to compute the B-spline basis functions in both u and v directions with the u and v arguments passed in. According to NURBS surface formula introduced in the NURBS Theory section, the weight values are multiplied by their corresponding basis functions and control points in both u and v directions.

```
void get_surface_vertex(vec3 &vertex, float u, float v)
{
    get_basis_functions(degree_u, u, knots_u, basis_u);
    get_basis_functions(degree_v, v, knots_v, basis_v);
    unsigned int u_count = basis_u.size(), v_count = basis_v.size();
    int index = 0;
    float factor = 0;
    for(unsigned int i = 0; i < u_count; i++)
    {
        float temp = 0;
        for(unsigned int j = 0; j < v_count; j++)
        {
            temp += basis_u[j] * weights[index++];
        }
        factor += temp * basis_v[i];
    }
    factor = 1 / factor;
    index = 0;
    for(unsigned int i = 0; i < v_count; i++)
    {
        vertices_v[i] = vec3(0, 0, 0);
        for(unsigned int j = 0; j < u_count; j++)
        {
            vertices_v[i] += basis_u[j] * weights[index] *
ctrl_points[index];
            index++;
        }
    }
    vertex = vec3(0, 0, 0);
    for(unsigned int k = 0; k < v_count; k++)
    {
        vertex += basis_v[k] * vertices_v[k];
    }
    vertex = vertex * factor;
}
```

void add_knot_u(unsigned int index, float k)

void add_knot_v(unsigned int index, float k)

void set_knot_u(unsigned int index, float k)

void set_knot_v(unsigned int index, float k)

These fours functions are invoked to add new knot value to a NURBS object and set the value of existing knot in u and v direction.

void add_weight(float w)

void increase_weight_value(unsigned int index, float w)

These two functions are used to add new weight value to a NURBS object and adjust the value of existing weight.

void set_degree_u(int d)

void set_degree_v(int d)

These two function set the degree of an NURBS object.

void add_ctrl_points(const vec3 &v)

void set_ctrl_points(unsigned int index, const vec3 &v)

These two function add new control point and set the position of an control point of a NURBS object.

void reset()

This function clears all member variables in the class.

class sky_box

sky_box()

This is the constructor of class sky_box, which constructs the vertices of the 6 faces of the skybox according to the variable of the skybox size with uv coordinates attached.

void set_position(const vec3 &pos)

This function sets the central position of the skybox. The position information is stored in the translation matrix of the skybox.

void init(char *path)

This function initializes the texture shader of the skybox and loads the skybox texture.

void render(const mat4t &modelToProjection, int sampler)

This is the rendering function of the skybox.

class camera_control

void generate_matrix()

The function generates inverse of the view transformation matrix according to the vertical and horizontal viewing angles, the position of the viewing target and the viewing distance.

void add_view_distance(float f)

void set_view_distance(float distance)

These two functions help change the view distance of the camera.

void set_view_position(const vec3 &pos)

This function sets target position the camera is looking at.

void rotate_v(float f)

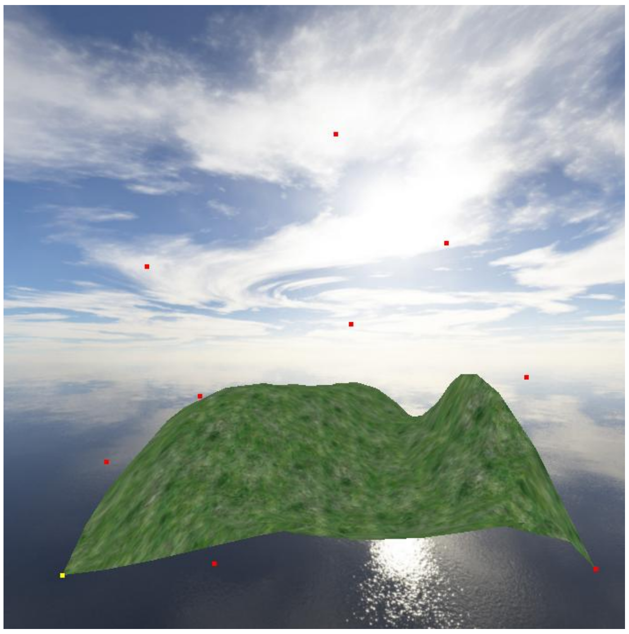
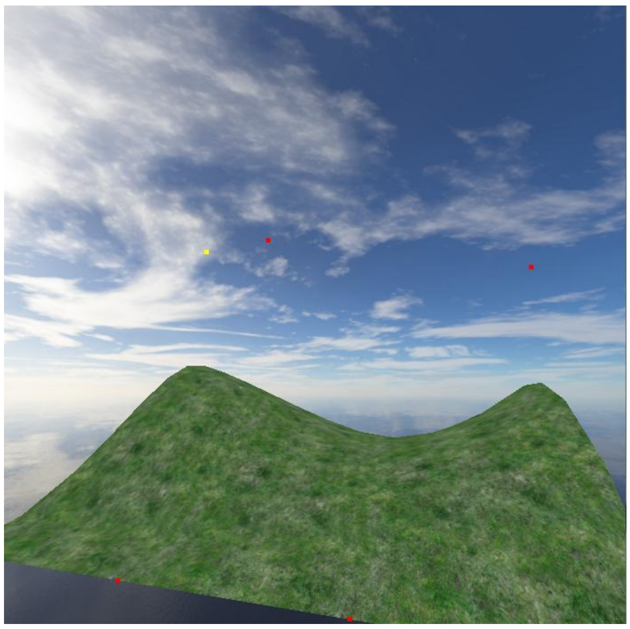
void rotate_h(float f)

These two functions rotate the camera vertically and horizontally.

const mat4t &get_matrix()

This function returns the inverse of the view matrix.

Screenshots of terrain interactively generated

class Water_Square

The "Water_Square" class is a base class to support the drawing of a single quad mesh. The class consists of an array to store the 12 vertices as each point of the quad contains 3 float values to represent the X, Y and Z axis.

void Render()

Inside the render function, it takes in the array of vertices and calls the OpenGL "glDrawArrays" method in "GL_QUADS" draw mode to render each sub section of the water plane.

void setP0(vec3 &value), void setP1(vec3 &value),

void setP2(vec3 &value), void setP3(vec3 &value)

These are the setter methods for each of the vertices; it takes in the vector 3 position as its parameter and assign it to the vertices array.

vec3 &getP1(),vec3 &getP1(),vec3 &getP1(),vec3 &getP1()

These are the getter methods to return the vector 3 position of the corresponding vertices.

class Water_Mesh_app

The "Water_Mesh_app" class is the main class that handles the camera controls and specific key presses for the manipulation of the water plane. It also contains helper functions to calculate the core elements that were mentioned in the theory section.

void app_init()

This method is automatically called once OpenGL is initialized, it handles the initialization of the variables which includes the color shader, different view matrix for the camera to world and model to world, wave parameters such as the wave length, amplitude, speed, wave steepness, directions and finally the 1 dimensional array that holds the position of each sub quad to form the water plane.

void draw_world(int x, int y, int w, int h)

This is the main drawing function that loops continuously throughout the execution of the program, it is where the key press function and the calculation of each vertex take place. A good optimization for this class would be rather letting the CPU to process each of the changing vertex, it could be sent to the vertex shader for the GPU to process vertex calculation instead.

void setCameraToWorld(float X, float Y, Z)

This is a helper function for the camera controls. It setup the identity matrix before applying the translation of the camera to world view.

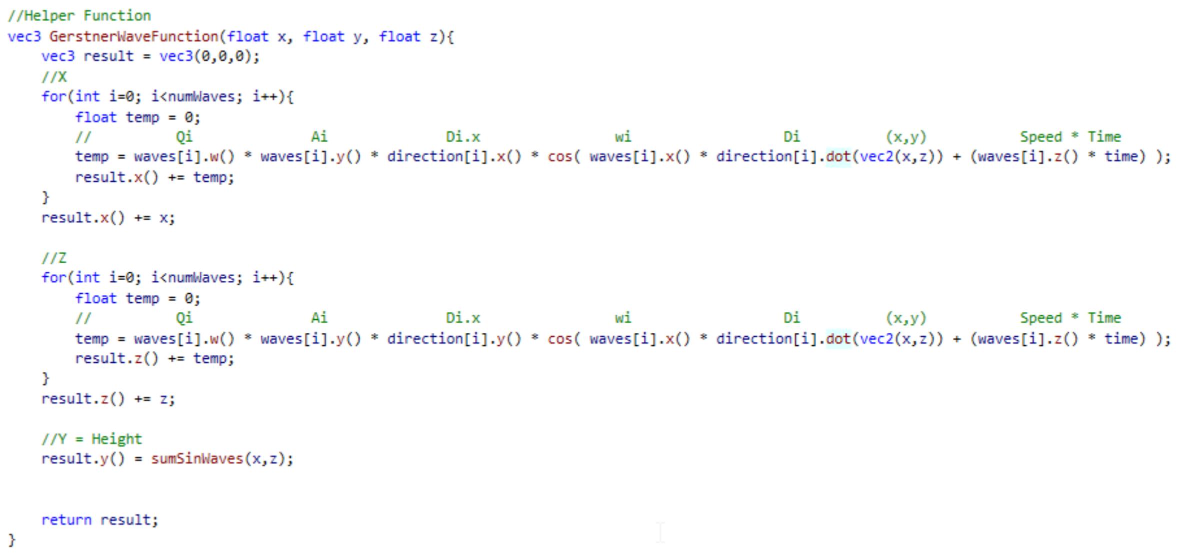
void simulate()

This is the main helper function that handles all the related key press to manipulate the water plane and camera controls. Here are each of the key presses:

- [U, J]: “U” increment the wave steepness factor and “J” have the vice versa effect.
- [I, K]: “I” increment the wave length in relation to frequency and “K” have the vice versa effect.
- [O, L]: “O” increment the amplitude factor of the wave and “L” have the vice versa effect.
- [M]: “M” increases the speed factor of the wave.
- [0, 9, 8]: Each of these key presses will set a specific wave parameters to visually output a specific wave result.
- [5, 6]: “5” display each parameter of the 4 waves, “6” display the sum of each wave parameters of the 4 waves.

vec3 GerstnerWaveFunction(float x, float y, float z)

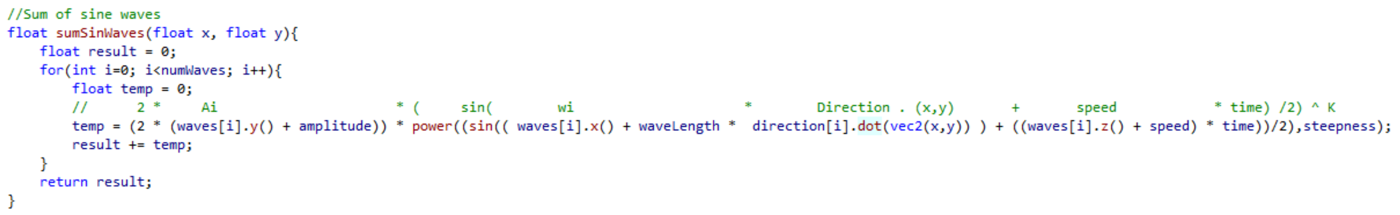
This is the method for calculating the Gerstner wave formula; it returns a vector 3 as its result.



```
//Helper Function
vec3 GerstnerWaveFunction(float x, float y, float z){
    vec3 result = vec3(0,0,0);
    //X
    for(int i=0; i<numWaves; i++){
        float temp = 0;
        //          Qi              Ai              Di.x                wi                  Di          (x,y)           Speed * Time
        temp = waves[i].w() * waves[i].y() * direction[i].x() * cos( waves[i].x() * direction[i].dot(vec2(x,z)) + (waves[i].z() * time) );
        result.x() += temp;
    }
    result.x() += x;

    //Z
    for(int i=0; i<numWaves; i++){
        float temp = 0;
        //          Qi              Ai              Di.x                wi                  Di          (x,y)           Speed * Time
        temp = waves[i].w() * waves[i].y() * direction[i].y() * cos( waves[i].x() * direction[i].dot(vec2(x,z)) + (waves[i].z() * time) );
        result.z() += temp;
    }
    result.z() += z;

    //Y = Height
    result.y() = sumSinWaves(x,z);


    return result;
}
```

float sumSineWaves(float x, float y)

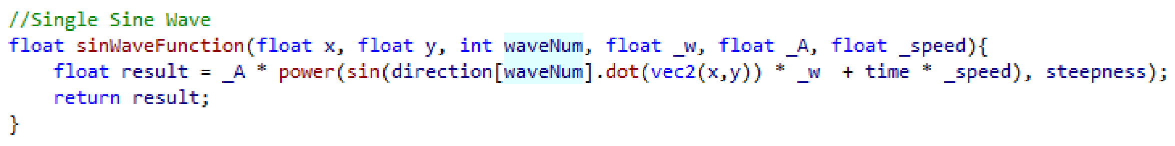
This is the method to calculate the summation of the sine waves with the consideration of the steepness factor by offsetting the sine wave result to be nonnegative and increase by an exponent of “k” to define the sharper and wider troughs of the wave.



```
//Sum of sine waves
float sumSinWaves(float x, float y){
    float result = 0;
    for(int i=0; i<numWaves; i++){
        float temp = 0;
        //      2 *     Ai                          * (     sin(        wi                      *       Direction . (x,y)       +       speed               * time) /2) ^ K
        temp = (2 * (waves[i].y() + amplitude)) * power((sin(( waves[i].x() + waveLength *  direction[i].dot(vec2(x,y)) ) + ((waves[i].z() + speed) * time))/2),steepness);
        result += temp;
    }
    return result;
}
```

float sinWaveFunction (float x, float y, int waveNum, float _w, float _A, float _speed)

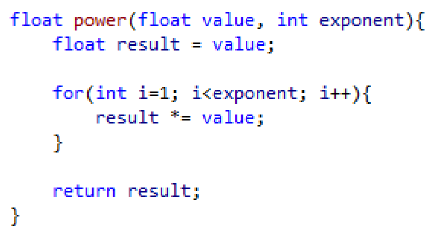
This function is purely for laying out the different wave parameter for early stages of implementing a single sine wave.



```
//Single Sine Wave
float sinWaveFunction(float x, float y, int waveNum, float _w, float _A, float _speed){
    float result = _A * power(sin(direction[waveNum].dot(vec2(x,y)) * _w  + time * _speed), steepness);
    return result;
}
```

float power(float value, int exponent)

This is a simple utility function for the power operation in maths, since the Octet framework did not provide one.

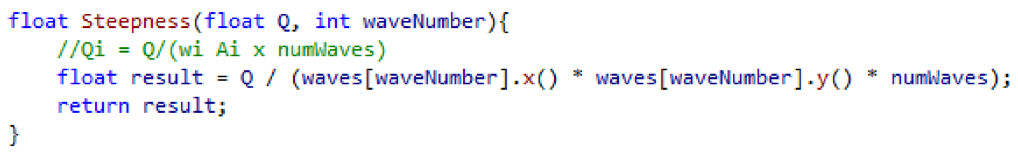


```
float power(float value, int exponent){
    float result = value;

    for(int i=1; i<exponent; i++){
        result *= value;
    }

    return result;
}
```

float Steepness(float Q, int waveNumber)

This function returns the steepness value of the wave by using a specific formula that take in consideration to vary from very smooth waves to sharpest wave.

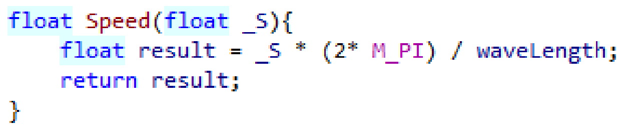
Formula: $Q_i = Q/(w_i\, A_i \text{ x } numWaves)$



```
float Steepness(float Q, int waveNumber){
    //Qi = Q/(wi Ai x numWaves)
    float result = Q / (waves[waveNumber].x() * waves[waveNumber].y() * numWaves);
    return result;
}
```

float Speed(float _S)

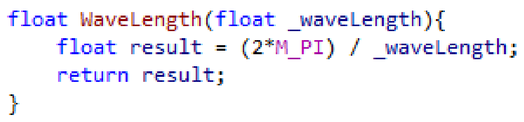
This function returns the distance the waves moves forward per second. It is expressed as the following: Phase-Constant = $S \text{ x } 2\pi/L$.



```
float Speed(float _S){
    float result = _S * (2* M_PI) / waveLength;
    return result;
}
```

float WaveLength(float _waveLength)

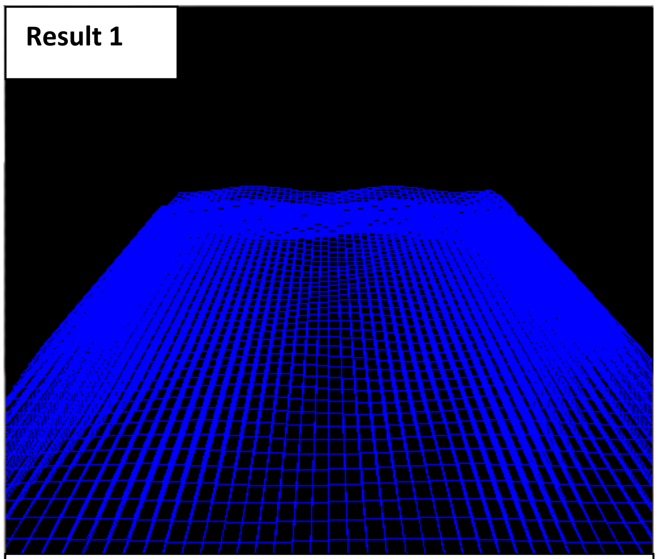
This function returns the wave to wave distance between each crest in world space. It relates to frequency W. Formula: $w = 2\pi/L.$



```
float WaveLength(float _waveLength){
    float result = (2*M_PI) / _waveLength;
    return result;
}
```

Sample waves with different parameters:

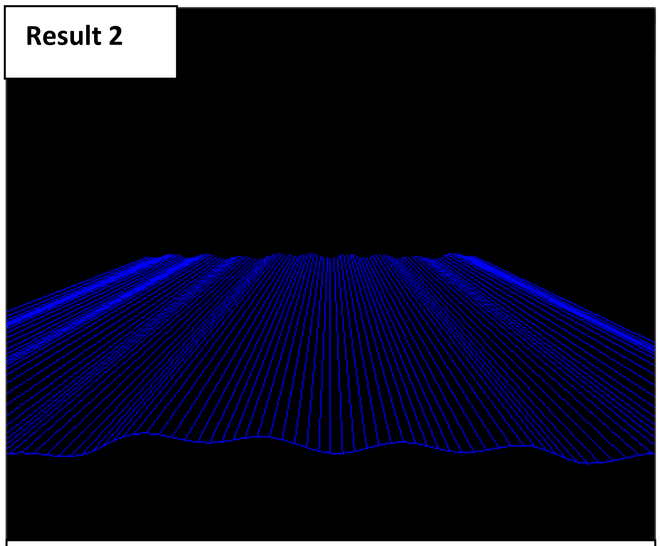
Result 1



Default Wave Setup:

- WaveLength (w): 1
- Amplitude (A): 2
- Speed (S): 2.5
- Steepness: 7

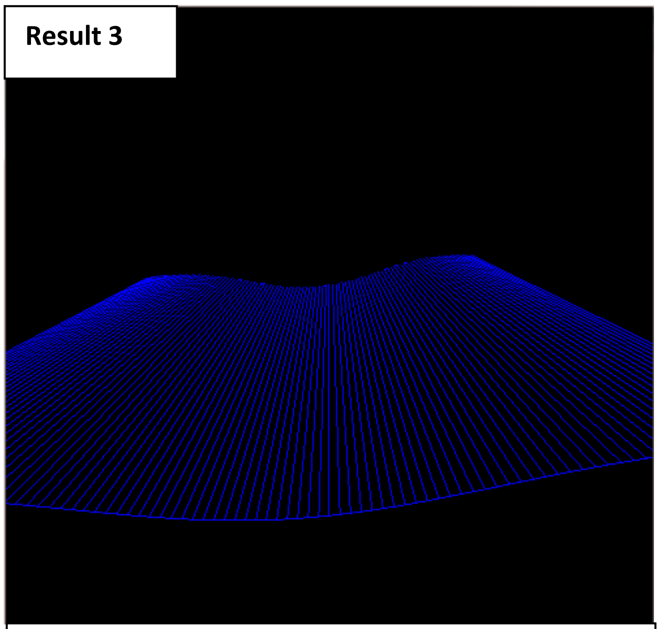
Result 2



Small ripple Wave Setup:

- WaveLength (w): 3.28
- Amplitude (A): 100
- Speed (S): 3.06
- Steepness: 24.6

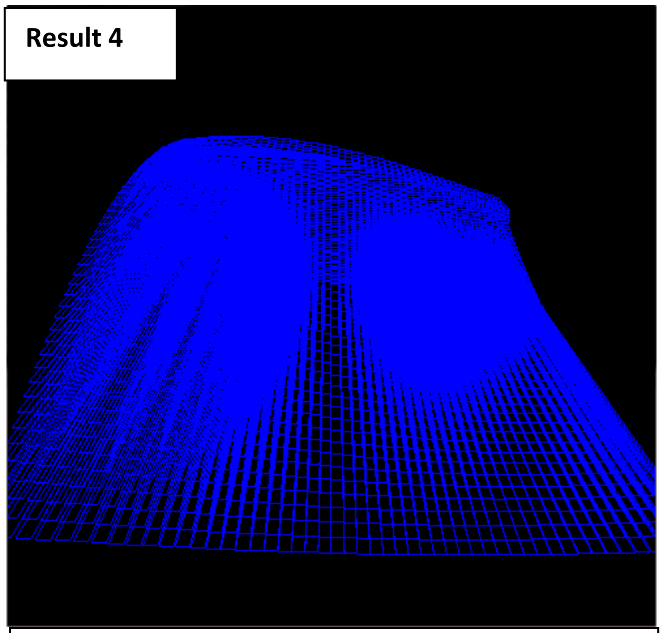
Result 3



Smooth Calm Sea Wave Setup:

- WaveLength (w): 3.04
- Amplitude (A): 20
- Speed (S): 10.1
- Steepness: 12.59

Result 4



Stormy Rough Wave Setup:

- WaveLength (w): 3.24
- Amplitude (A): 20
- Speed (S): 10.1
- Steepness: 12.59

Conclusion

During the implementation of the Gerstner wave function, it has been tested and successfully implemented into code, please refer to the result section. In result 1, it clearly shows the characteristic of Gerstner wave where by the sharper crests are formed by moving vertices toward each crest.

Summation of Sine Wave verse Gerstner Wave

The Gerstner wave formula is an extension of the summation of sine wave since they both use the same parameters to control the waves and for the height variance they both use the summation of sine waves. However with the approach of summation of sine wave, it uses the exponent function to give the sharp looking wave where as Gerstner wave calculates each of the X and Z plane to form sharper crest hence why there are high density of vertices at the curving of the wave.

Bibliography

- CS3621 Introduction to Computing with Geometry Course Notes. 2014.*CS3621 Introduction to Computing with Geometry Course Notes*. [ONLINE] Available at: http://www.cs.mtu.edu/~shene/COURSES/cs3621/NOTES/. [Accessed 11 April 2014].
- Gamasutra - Using NURBS Surfaces in Real-Time Applications. 2014.*Gamasutra - Using NURBS Surfaces in Real-Time Applications*. [ONLINE] Available at:http://www.gamasutra.com/view/feature/131808/using_nurbs_surfaces_in_realtime_.php?print=1. [Accessed 11 April 2014].
- Non-uniform rational B-spline - Wikipedia, the free encyclopedia. 2014. *Non-uniform rational B-spline - Wikipedia, the free encyclopedia*. [ONLINE] Available at: http://en.wikipedia.org/wiki/Non-uniform_rational_B-spline. [Accessed 11 April 2014].
- 5. B-Spline Curves . 2014. *5. B-Spline Curves* . [ONLINE] Available at:http://www.doc.ic.ac.uk/~dfg/AndysSplineTutorial/BSplines.html. [Accessed 11 April 2014].
- GPU Gems – Chapter 1. Effective Water Simulation from Physical Models. 2014. GPU Gems – Chapter 1. Effective Water Simulation from Physical Models. [ONLINE] Available at: http://http.developer.nvidia.com/GPUGems/gpugems_ch01.html [Accessed 31 March 2014].
- Fast Fourier transform - Wikipedia, the free encyclopedia. 2014. *Fast Fourier transform - Wikipedia, the free encyclopedia*. [ONLINE] Available at:http://en.wikipedia.org/wiki/Fast_Fourier_transform. [Accessed 11 April 2014].