Synthétisation d'une image numérique

- Préambule : NH90 VMT

Synthétisation d'une image numérique

- I / Rendu graphique
- II / Champ de hauteur

Optimalité :

Choix, contraintes, hasards.

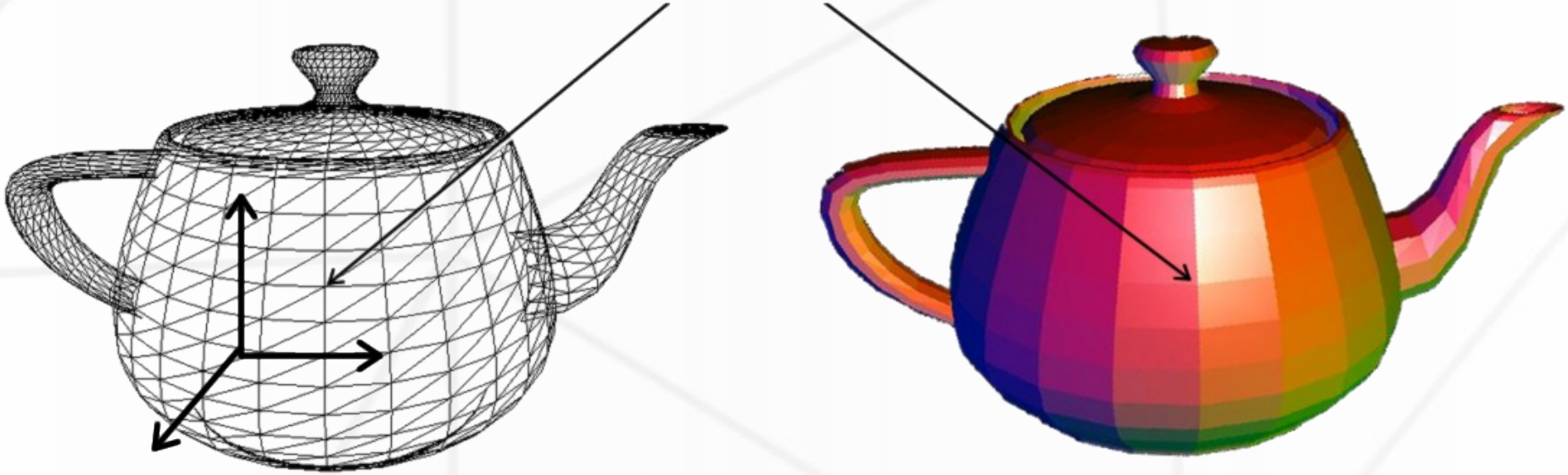
Rendu graphique

- Objet numérique

$$L=((V_1,V_2,V_3),(V_1,V_2,V_4),\ldots,(V_i,V_j,V_k))$$

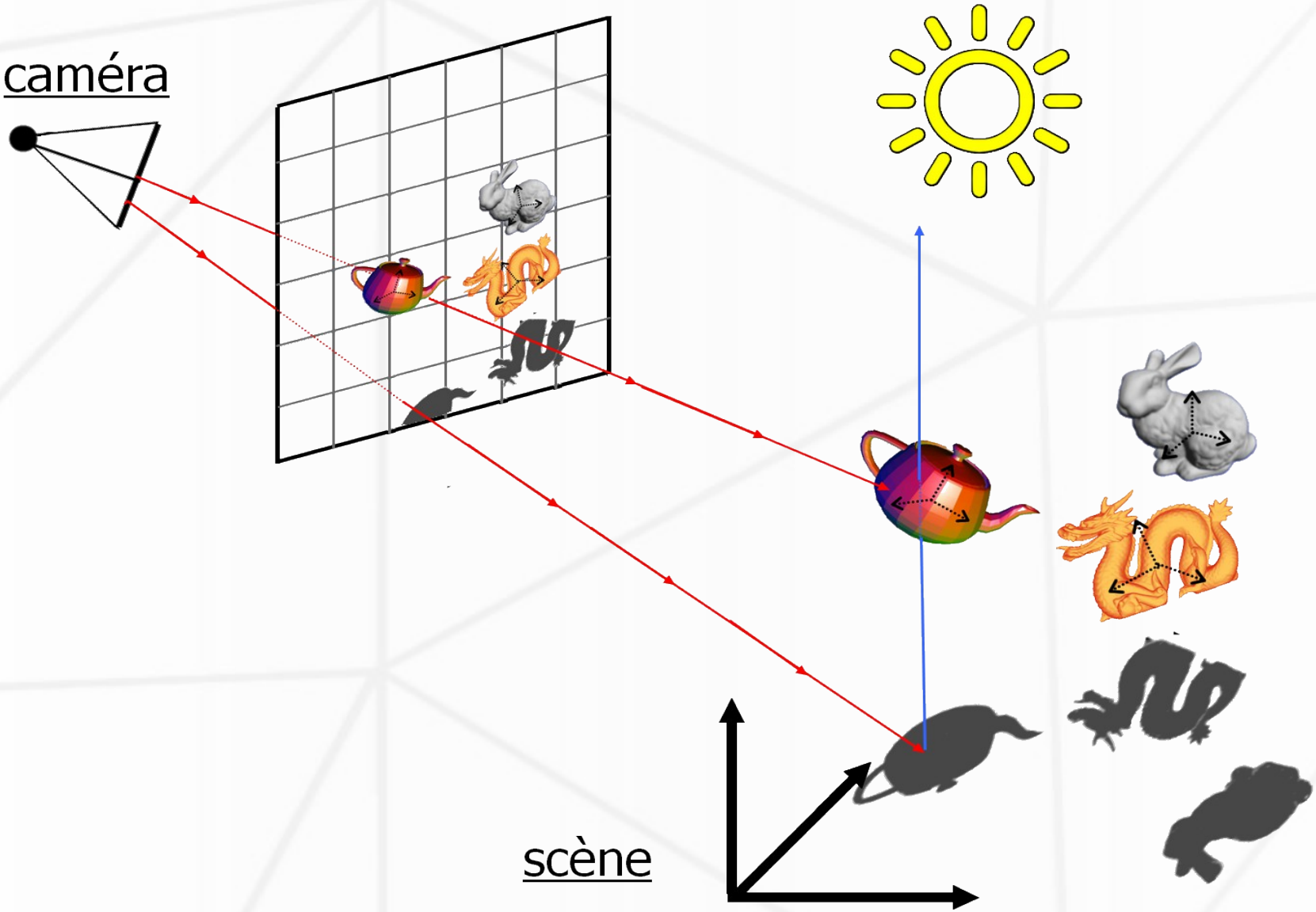
$$V_i=((x_i,y_i,z_i),(r_i,g_i,b_i),\ldots)$$

$$V_i=((1.4,2.8,1.1),(255,80,100),\ldots)$$



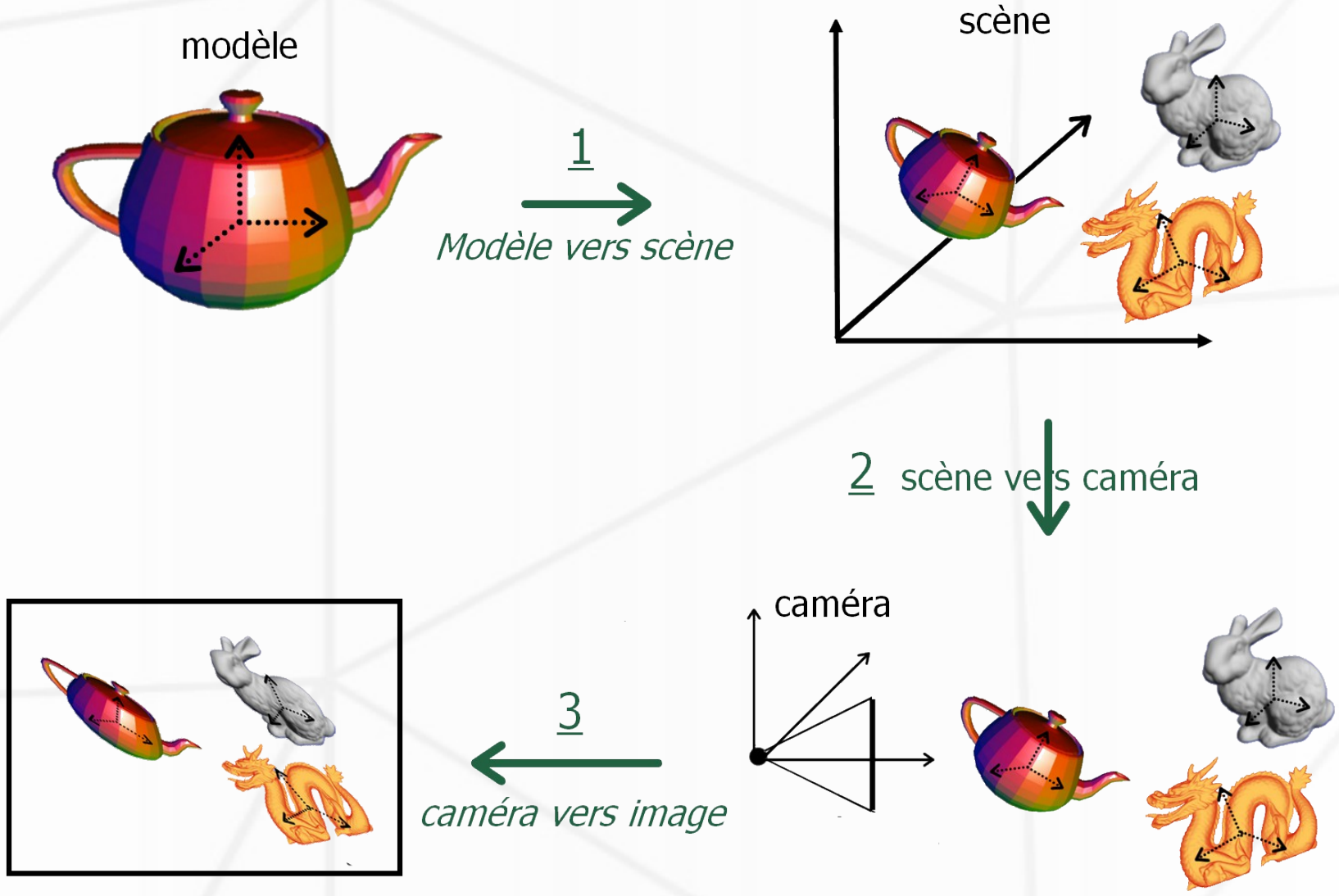
Rendu graphique

- Lancer de rayons



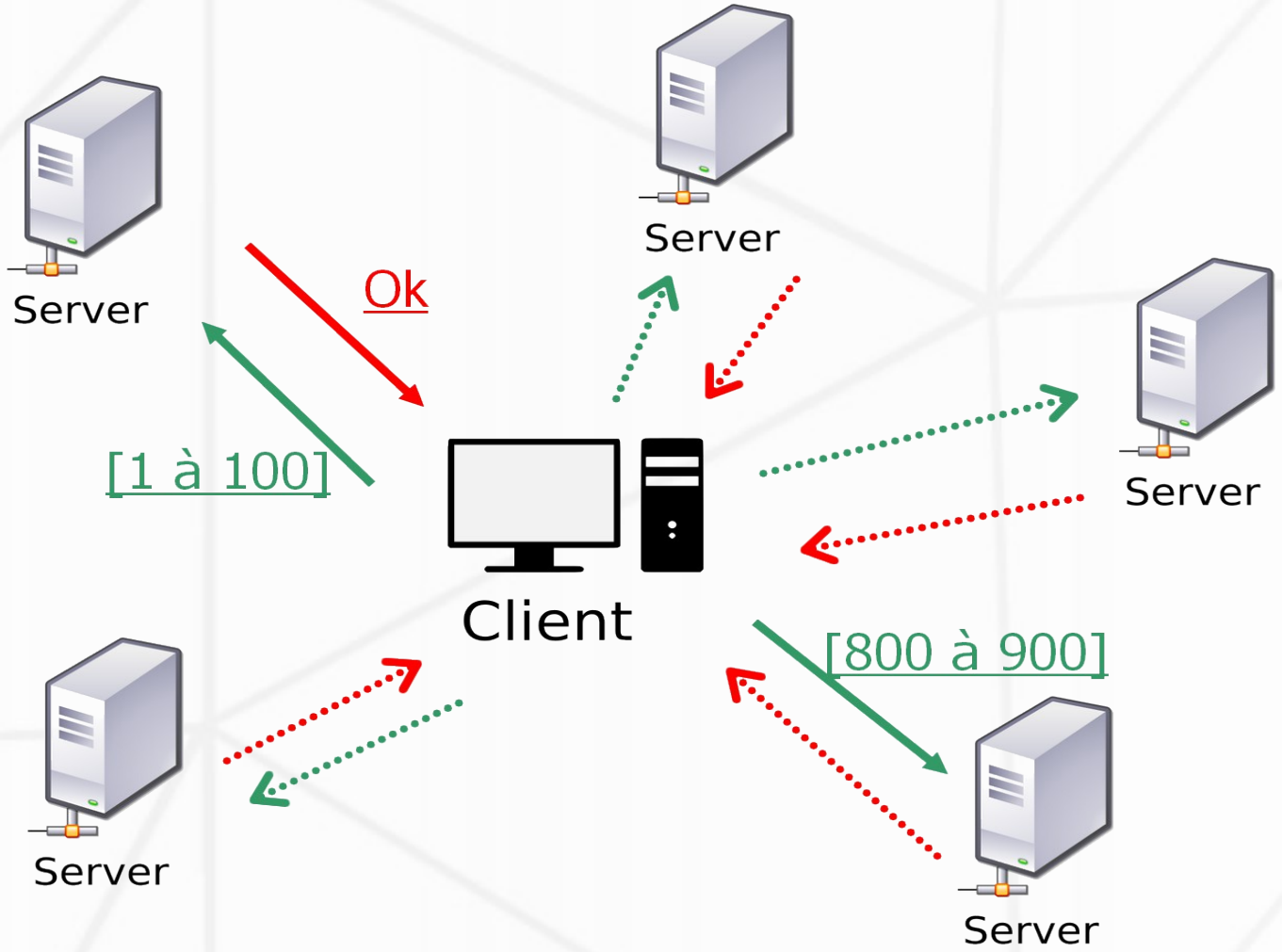
Rendu graphique

- Rastérisation



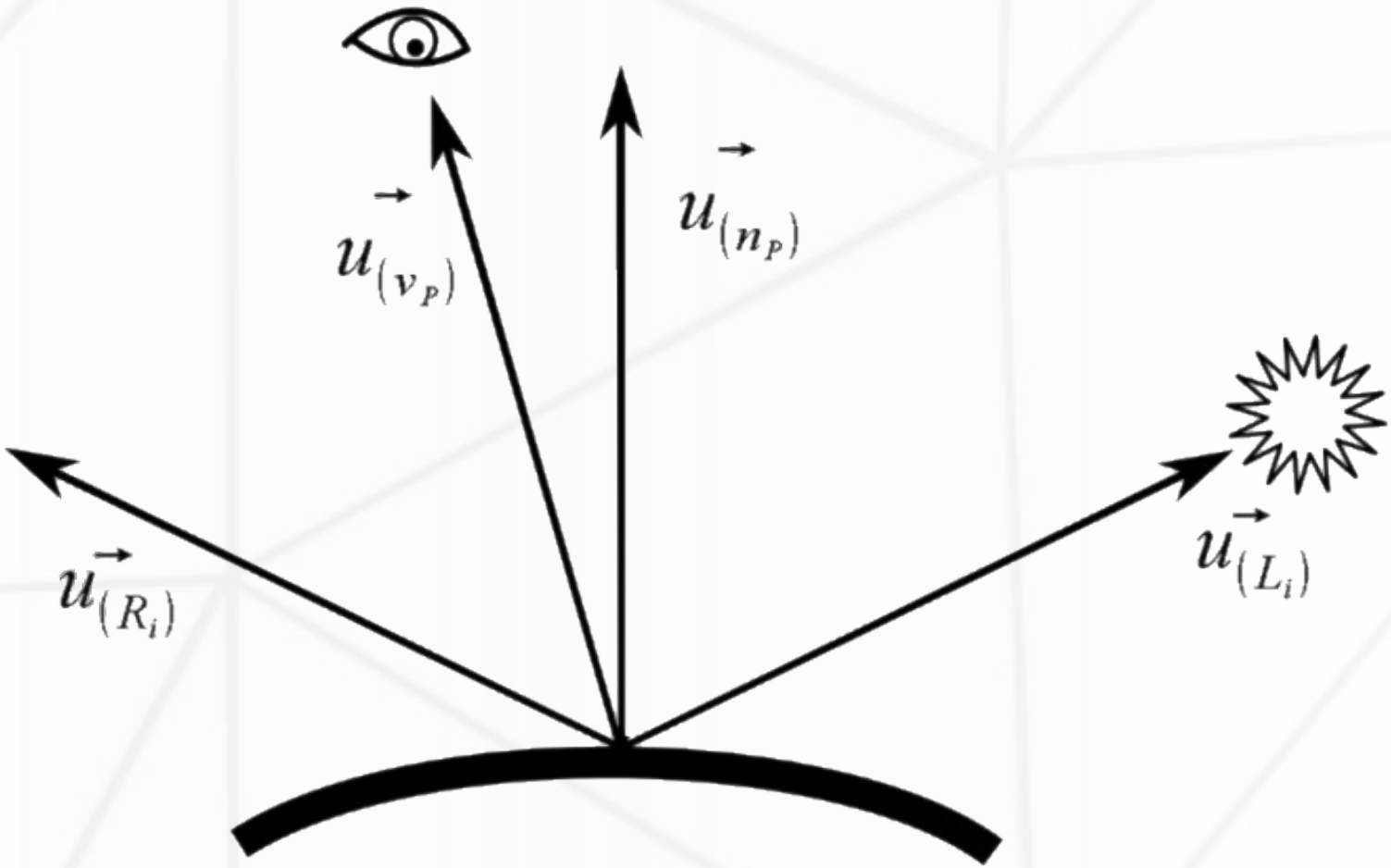
Rendu graphique

- Parallélisation



Eclairage : ombrage de Phong

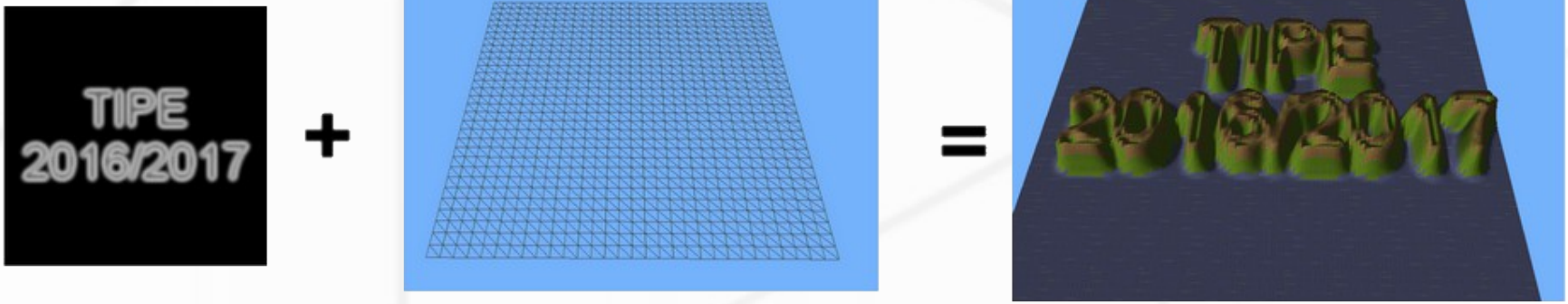
$$I(P) = I_A(P) + \sum_{i=1}^{n} \vec{u}_{(L_i)} \bullet \vec{u}_{(n_P)} * I_i + (\vec{u}_{(R_i)} \bullet (\vec{u}_{(v_P)}))^{\alpha} * I_i$$

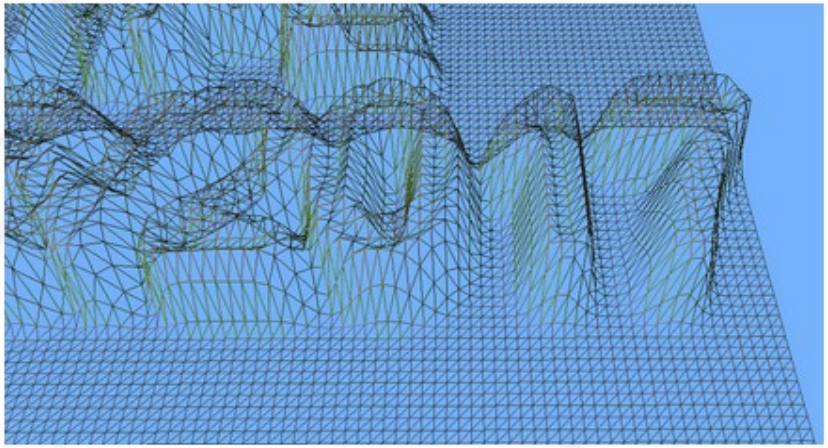


Champ de hauteur

- Définition

$$H : \mathbb{R}^2 \to \mathbb{R}$$
$$(x, z) \to H(x, z) = y$$

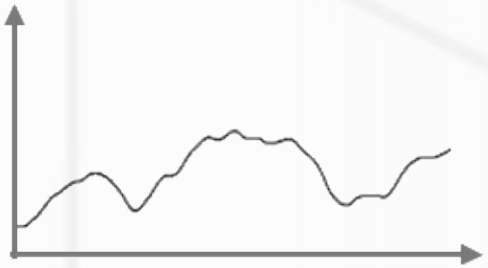


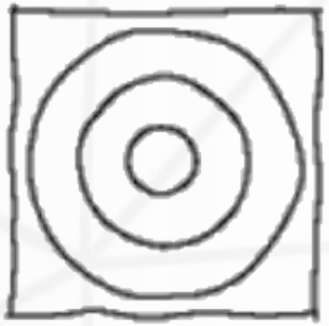


Champ de hauteur

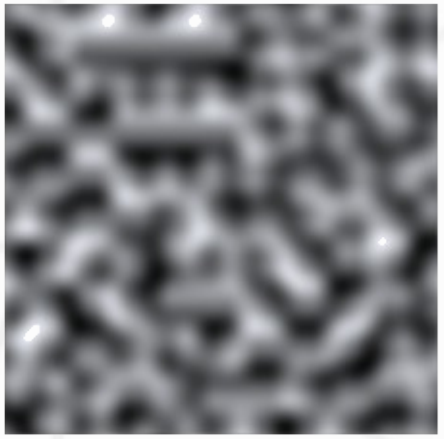
- Génération procédural : fonctions de bruits

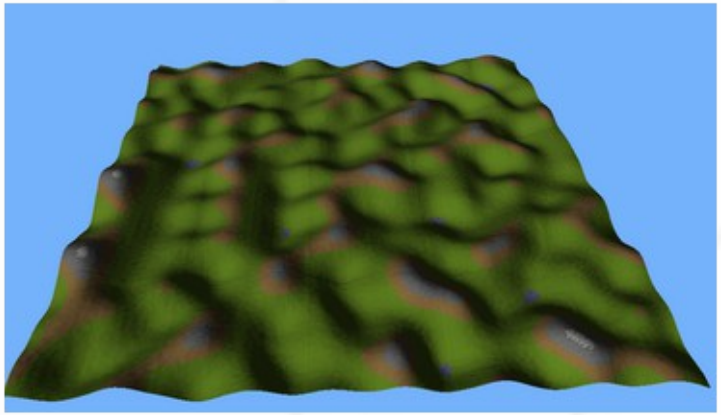
F(t) = x





F(x, y) = z

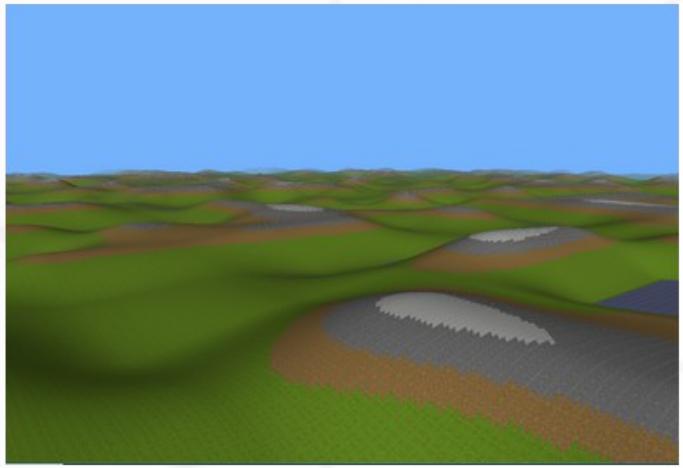




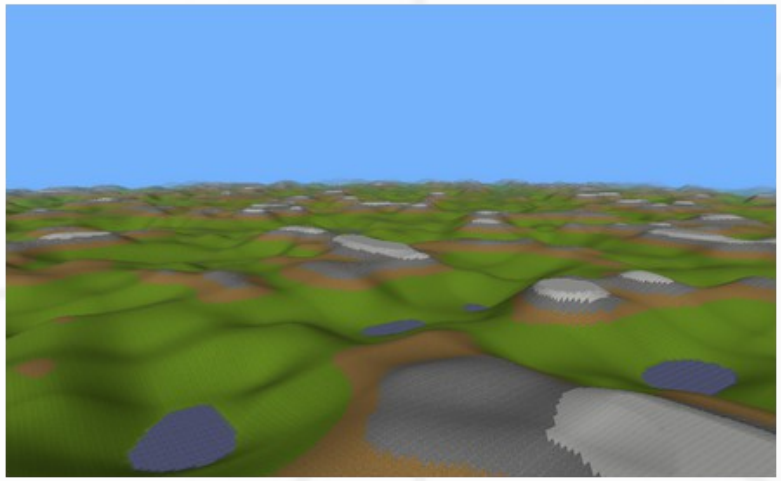
Champ de hauteur

- Génération procédural : combinaisons

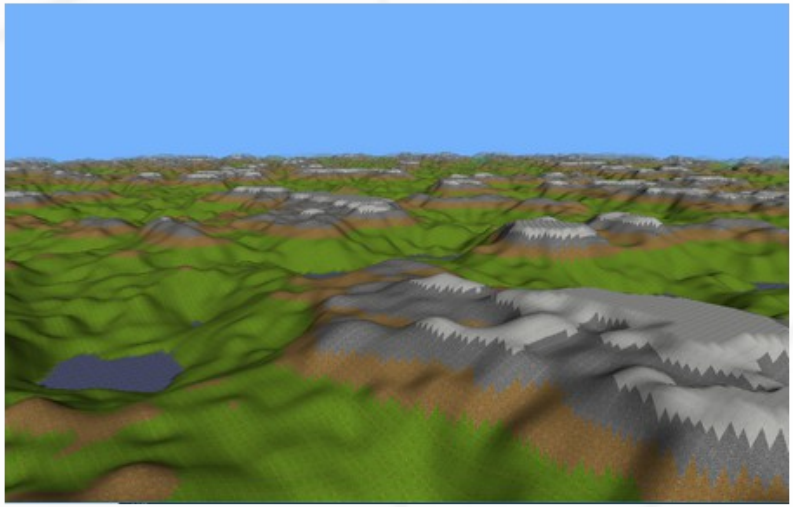
```
x, y
hauteur = 0
Pour chaque octave dans octaves:
    f = octave.frequence
    I = octave.amplitude
    h = h + F(x * f, y * f) * I
```



1 octave



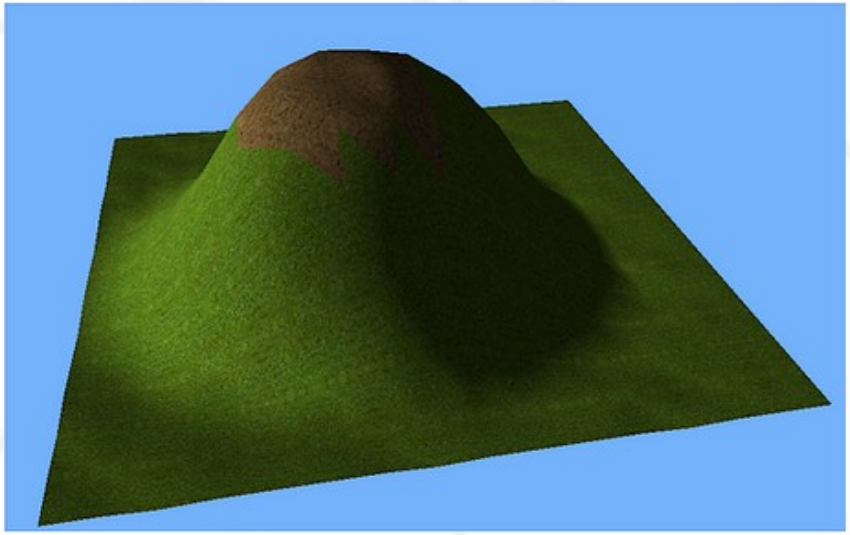
2 octaves

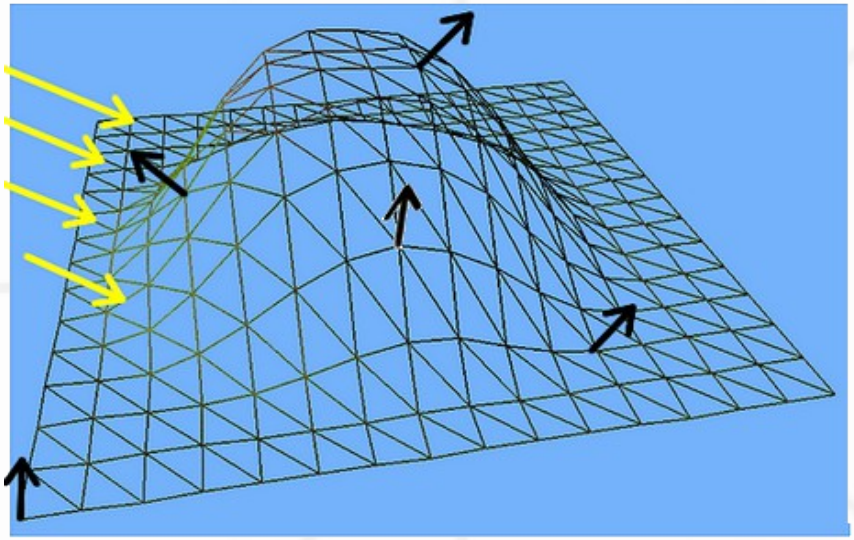


4 octaves

Champ de hauteur

- Calcul des normales





$$F(x,y,z) = H(x,z) - y$$

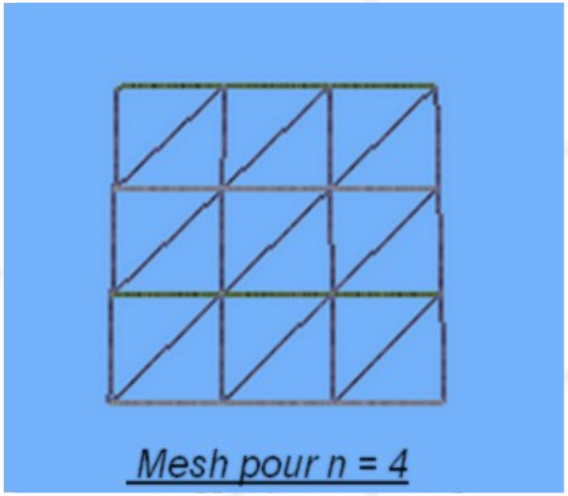
$$\vec{N}(x_0,y_0,z_0) = \begin{vmatrix} \frac{\partial F(x_0,y_0,z_0)}{\partial x} \\ \frac{\partial F(x_0,y_0,z_0)}{\partial y} \\ \frac{\partial F(x_0,y_0,z_0)}{\partial z} \end{vmatrix}$$

$$= \begin{vmatrix} \frac{H(x_0+dx,z_0) - H(x_0,z_0)}{dx} \\ -1 \\ \frac{H(x_0,z_0+dz) - H(x_0,z_0)}{dz} \end{vmatrix}$$

Champ de hauteur

- Complexité spatiale

$$n = \text{'nombre de points sur un coté du terrain'}$$
$$M(n) = \text{'mémoire total utilisé'}$$
$$N(n) = \text{'nombre de vecteurs distincts'}$$
$$T(n) = \text{'nombre de triangles'}$$
$$I(n) = \text{'nombre d'indices du maillage'}$$



Mesh pour n = 4

$$S = (3+3+1) * 4 = 28\,octets$$
$$V = ((x, y, z), (n_x, n_y, n_z), c)$$

$$N(n) = n^2$$
$$T(n) = 2(n-1)^2$$
$$I(n) = 3T(n)$$

Champ de hauteur

- Complexité spatiale
  - Formats de stockage

$$L_1(n) = ((V_{1,}V_{2,}V_3)_{1,}(V_{1,}V_{2,}V_4)_{2,}\dots,(V_a,V_b,V_c)_{(T(n))})$$
$$M_1(n) = I(n) * S$$
$$= 168n^2 + o(n^2)$$

$$L_2 = (V_1, V_{2,}\dots, V_n)$$
$$I_2 = ((1,2,3)_{1,}(1,2,4)_{2,}\dots,(a,b,c)_{(T(n))})$$
$$M_2(n) = N(n) * S + 2 * I(n)$$
$$= 40n^2 + o(n^2)$$

Champ de hauteur

- Complexité spatiale

$$n_0 = \frac{\sqrt{(6 * S * (S - 2))} + 6 * (S - 2)}{5 * S - 12}$$

n	0	n_0	+oo
$M_2(n) - M_1(n)$	+	-	

$S = 28\,octets \Rightarrow n_0 = 2$

$n = 1024 \Rightarrow M_1(n) = 168\text{Mo}$

$M_2(n) = 40\text{Mo}$

$M_2(n) - M_1(n) = -\ 128\text{Mo}$

$T(n) = 2\ 000\ 000\ triangles$

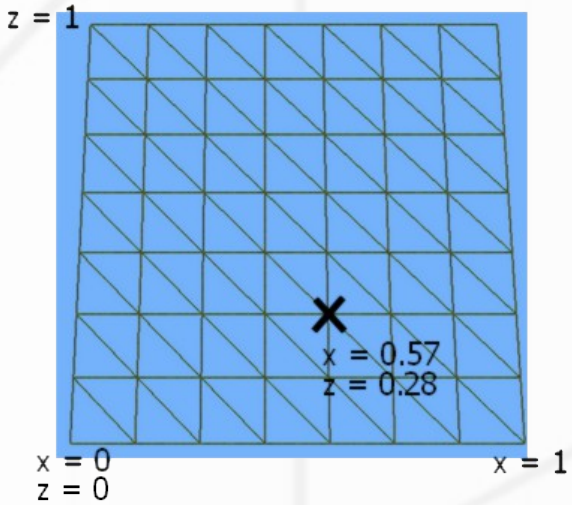
→ 2ème méthode

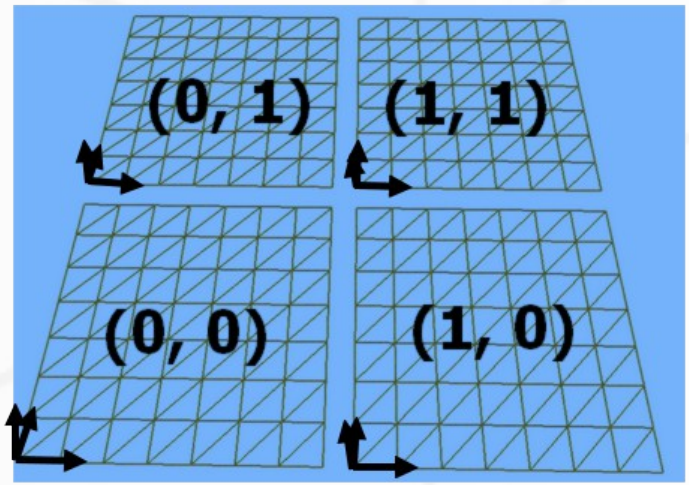
Champ de hauteur

- Complexité spatiale

$$((\cancel{x}, y, \cancel{z}), (n_x, \cancel{n_y}, n_z), c)$$

$$7 * 4 = 28\, octets \rightarrow 4 * 4 = 16\, octets$$



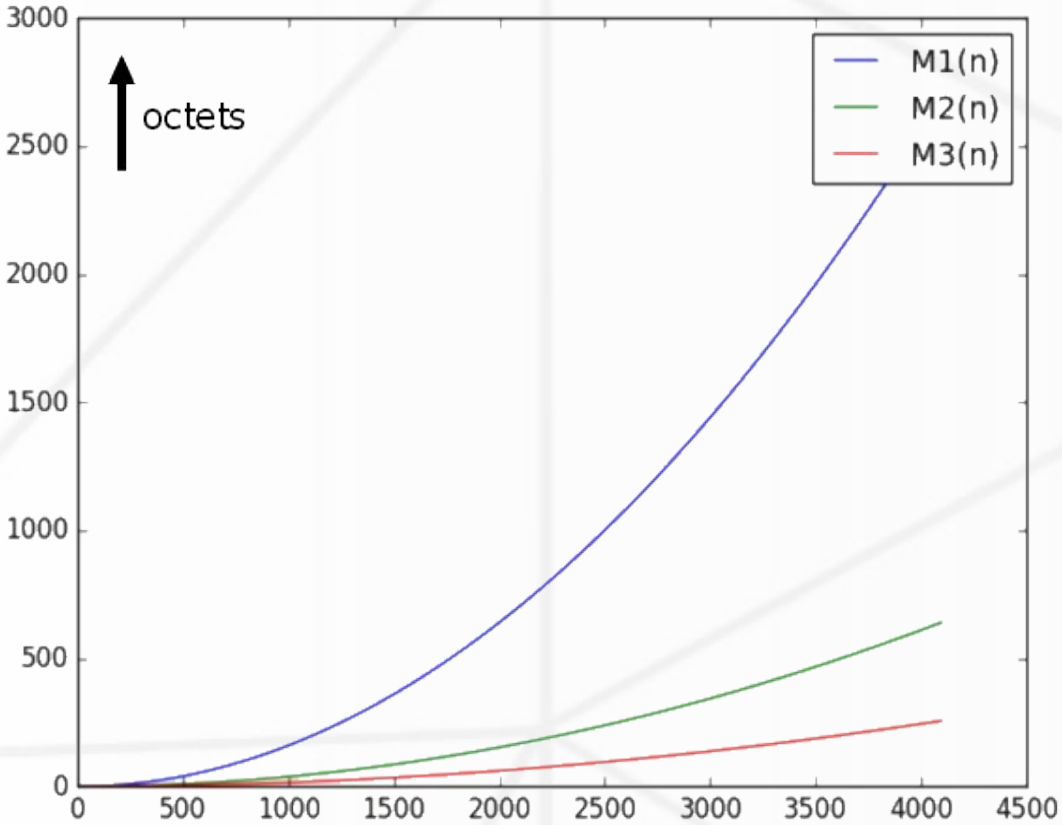


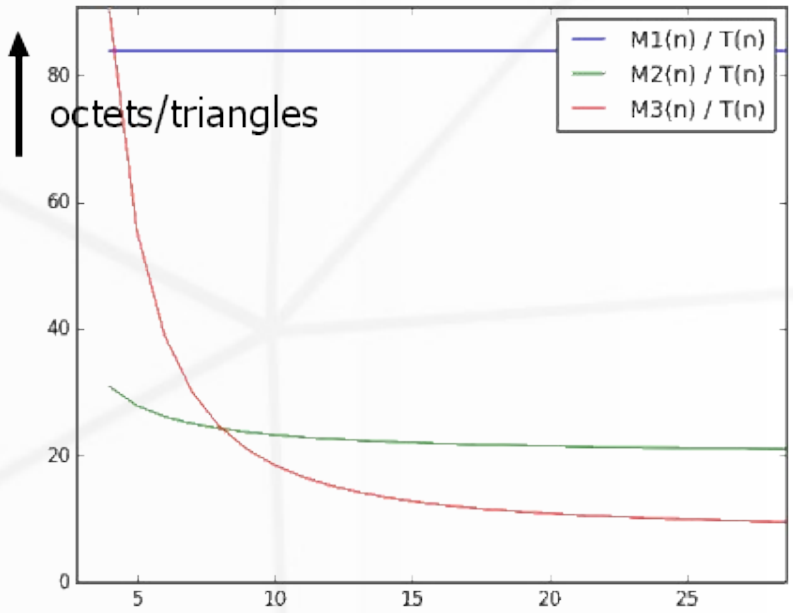
$$M_3 = N(n) * S + \frac{n}{n_u} * (16 * 4 + 2 * 4) + I(16) * 2$$

$$= 16n^2 + o(n^2)$$

Champ de hauteur

- Complexité spatiale : conclusion

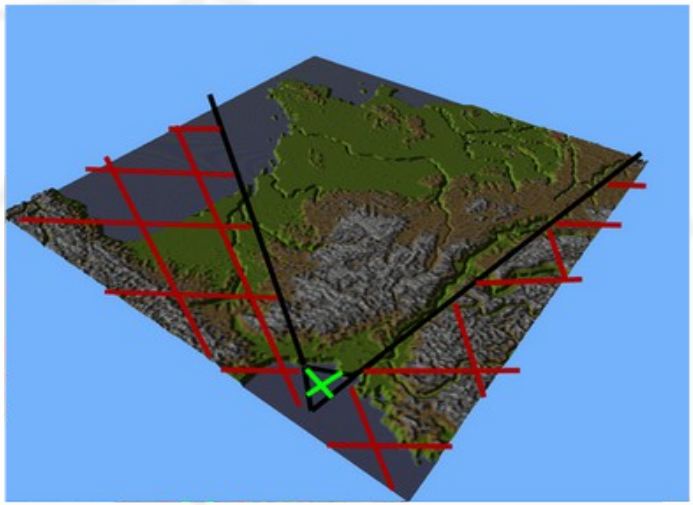




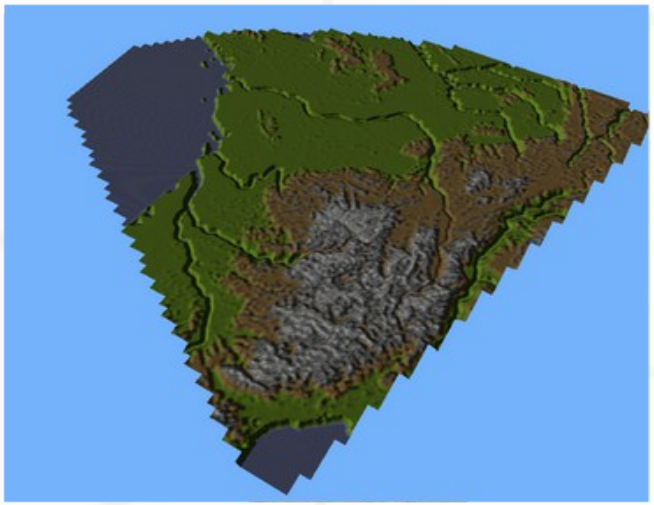
format	M_1	M_2	M_3
octet / vecteur	$S=28$	$S=28$	$S=16$
octets / triangles	84	20	8

Champ de hauteur

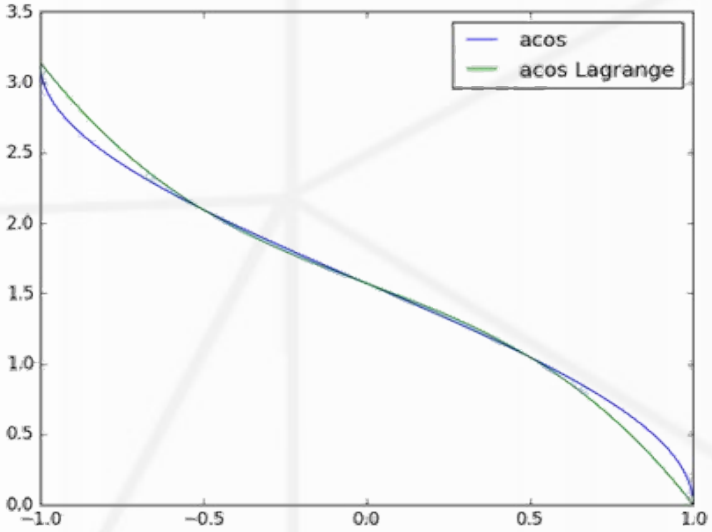
- Etude temporelle : culling

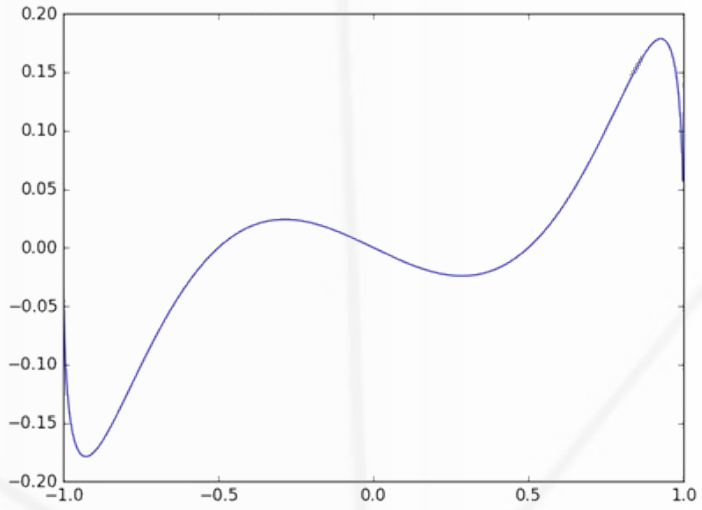


1024 terrains



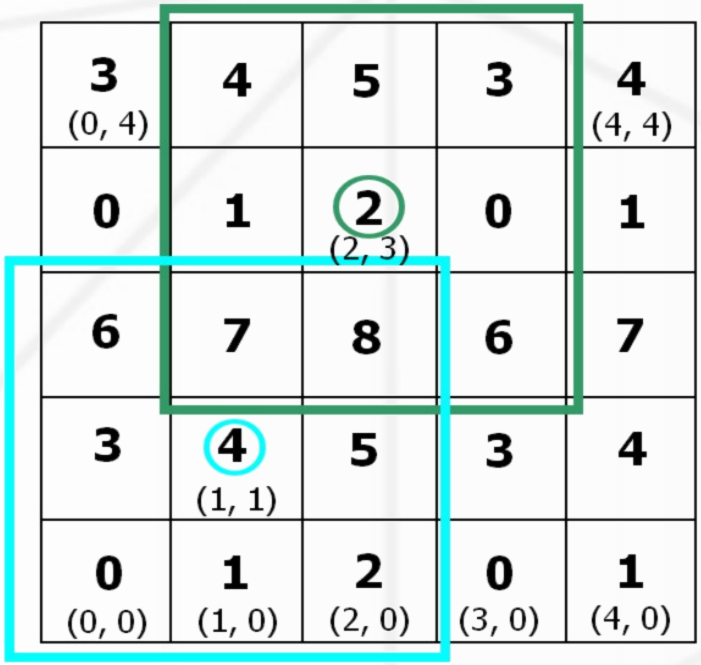
791 terrains





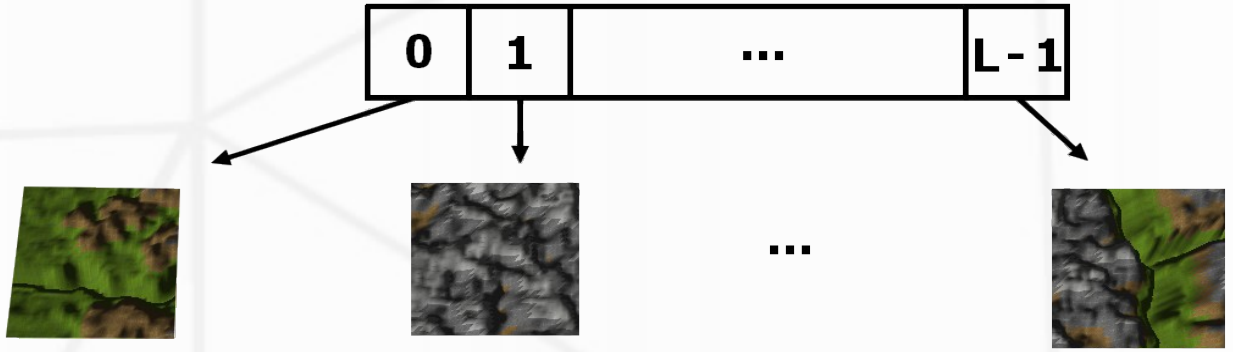
Champ de hauteur

- Etude temporelle : table de hashage

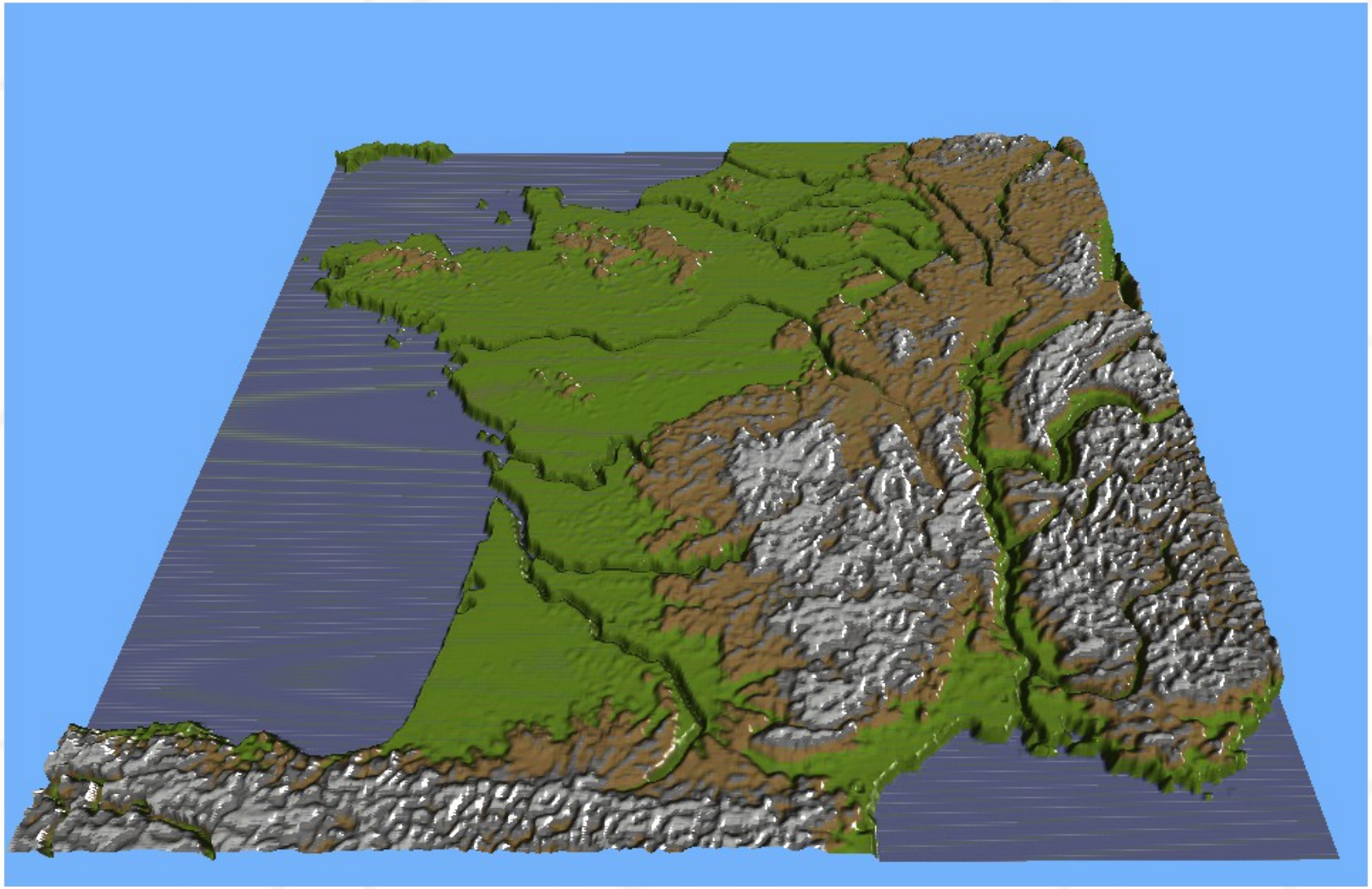


$$i : \mathbb{N}^2 \rightarrow [0; L^2[$$
$$(x, y) \rightarrow (x + y * L) \% L^2$$

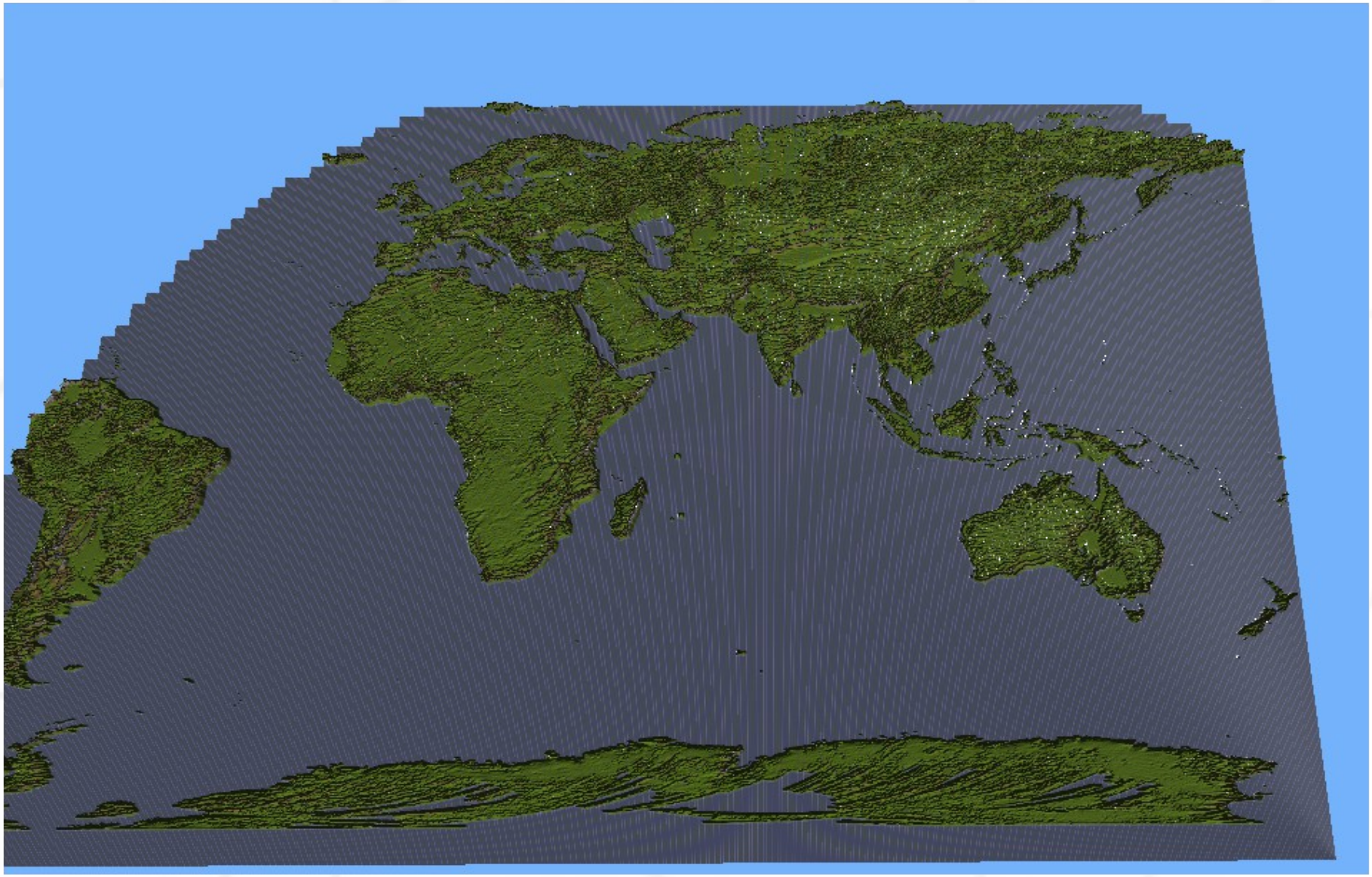
$$(ici, L=3)$$



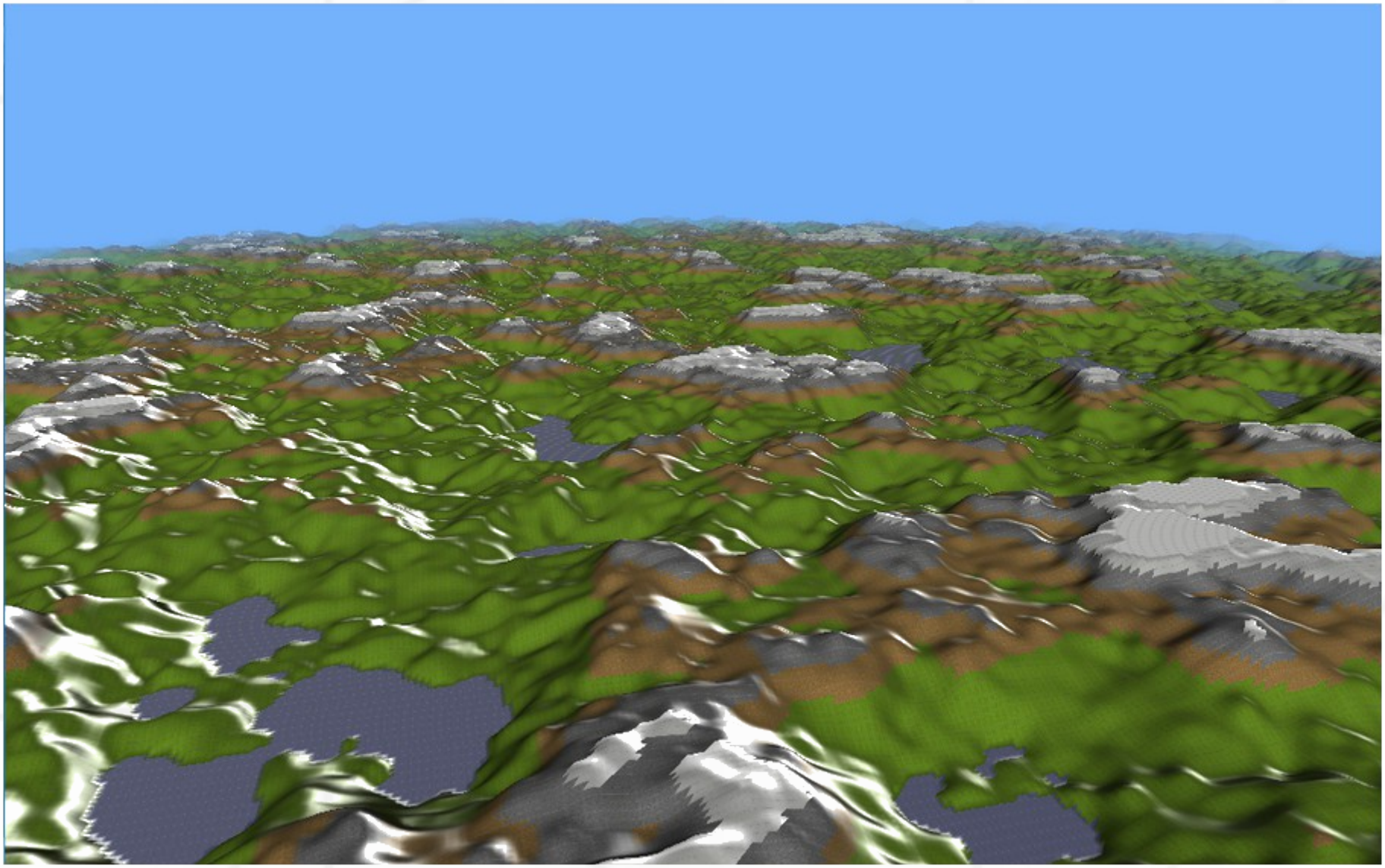
Illustrations



Illustrations



Illustrations



Illustrations

Sources

- [1] Wikipédia : Infographie : https://fr.wikipedia.org/wiki/Infographie
- [2] Wikipédia : Rastérisation : https://fr.wikipedia.org/wiki/Rasterisation
- [3] Wikipédia : Maillage triangulaire : https://fr.wikipedia.org/wiki/Mesh_(objet)
- [4] NVIDIA : Parralélisation : http://www.nvidia.com/object/what-is-gpu-computing.html
- [5] The University of Texas at Austin : Ombrage de Phong http://www.cs.utexas.edu/~bajaj/graphics2012/cs354/lectures/lect14.pdf
- [6] Wikipédia : Heightmaps : https://en.wikipedia.org/wiki/Heightmap
- [7] Amit PATEL, redblobgames : Génération procédurale : http://www.redblobgames.com/maps/terrain-from-noise/
- [8] Adrian BIAGIOLI : Bruit de Perlin : http://flafla2.github.io/2014/08/09/perlinnoise.html
- [9] Simulateur de vol : http://www.defense.gouv.fr/actualites/economie-et-technologie/la-dga-presente-le-1er-simulateur-europeen-de-formation-a-la-maintenance

Sommaire

1 Programme de rendu

renderer/shaders/terrain.vs

```
#version 330 core

in vec2   pos;
in vec2   uv;
in float  height;
in vec2   normal;
in int    textureID;

out float     visibility;
out vec3      pass_normal;
out vec2      pass_uv;
flat out int  pass_textureID;
out vec3      viewVec;

//view and projection matrix
uniform mat4 mvp_matrix;

//transformation matrix
uniform mat4 transf_matrix;

# define TERRAIN_SIZE (16.0)
# define TERRAIN_RENDER_DISTANCE (64)
# define RENDER_DISTANCE (TERRAIN_RENDER_DISTANCE * TERRAIN_SIZE)

void main(void) {

    vec4 world_pos = transf_matrix * vec4(pos.x, height, pos.y, 1.0);
    viewVec = normalize(-world_pos.xyz);
    gl_Position = mvp_matrix * world_pos;

    //fog calculation, visibility = pow(distance, 8)
    visibility = 0.0f;
    visibility = length(gl_Position.xz) / float(RENDER_DISTANCE);
    visibility = visibility * visibility;
    visibility = visibility * visibility;
    visibility = visibility * visibility;
    visibility = clamp(visibility, 0, 1);

    pass_normal = normalize(vec3(normal.x, 1.0, normal.y));
    pass_uv = uv;
    pass_textureID = textureID;
}
```

renderer/shaders/terrain.fs

```
#version 330 core

in float    visibility;
in vec3     pass_normal;
in vec2     pass_uv;
flat in int pass_textureID;
in vec3     viewVec;

out vec4 vertexColor;

uniform vec3 sky_color;
uniform vec3 sunray;

uniform sampler2D textureSampler;

# define TX_UNIT (1 / 5.0)
# define MIN_INTENSITY (0.2)
# define SPECULAR_DAMPING (4.0)
```

```
19
20  void main(void) {
21      //texture
22      float uvx = (pass_uv.x * TX_UNIT + pass_textureID * TX_UNIT);
23      float uvy = pass_uv.y;
24      vec3 txcolor = texture(textureSampler, vec2(uvx, uvy)).rgb;
25      vec3 color = txcolor;
26      //phong ligthing model
27      float intensity = max(dot(pass_normal, sunray), MIN_INTENSITY);
28      color *= intensity;
29
30      //specular lighting
31      vec3 reflectVec = reflect(-sunray, pass_normal);
32      float specAngle = max(dot(reflectVec, viewVec), 0.0);
33      float specular = pow(specAngle, SPECULAR_DAMPING);
34      color += specular * vec3(1.0, 1.0, 1.0);
35
36      //apply fog
37      vertexColor = vec4(mix(color, sky_color, visibility), 1.0);
38  }
```

renderer/includes/renderer.h

```
1   #ifndef RENDERER_H
2   # define RENDERER_H
3
4   # include "array_list.h"
5   # include "hmap.h"
6   # include "cmaths.h"
7   # include "vec.h"
8   # include "mat.h"
9   # include "glh.h"
10  # include "tinycthread.h"
11  # include "noise.h"
12  # include <string.h>
13  # include <unistd.h>
14  # include <string.h>
15  # include <time.h>
16  # include <fcntl.h>
17
18  /** the camera data structures */
19  typedef struct  s_camera {
20      t_vec3f pos; //camera world position
21      t_vec3f rot; //camera rotation toward (x, y, z) axis
22      t_vec3f vview; //view vector (direction where we are currently lookin at)
23      float   fov; //field of view
24      float   near_distance; //near plane distance
25      float   far_distance; //far plane distance
26      float   movespeed; // move speed
27      t_mat4f mview; //view matrix
28      t_mat4f mproj; //projection matrix
29      t_mat4f mviewproj; //projection matrix times view matrix
30      t_vec2i terrain_index; //current world terrain index of the camera
31  }               t_camera;
32
33  //terrain detail (number of vertex per line)
34  # define TERRAIN_DETAIL (16)
35  //terrain width (and height)
36  # define TERRAIN_SIZE (16)
37  # define TERRAIN_UNIT (TERRAIN_SIZE / (float)TERRAIN_DETAIL)
38  // number of terrain to render in term of distance
39  # define TERRAIN_RENDER_DISTANCE (64)
40  // distance where terrain are kept loaded in memory
41  # define TERRAIN_LOADED_DISTANCE (TERRAIN_RENDER_DISTANCE)
42  // distance where terrain are kept loaded in memory
43  # define TERRAIN_KEEP_LOADED_DISTANCE (TERRAIN_LOADED_DISTANCE)
44  # define MAX_NUMBER_OF_TERRAIN_LOADED (TERRAIN_KEEP_LOADED_DISTANCE *
        TERRAIN_KEEP_LOADED_DISTANCE * 2 * 2)
45  // number of floats per vertex
46  # define TERRAIN_VERTEX_SIZE ((1 + 2) * sizeof(float) + 1 * sizeof(int))
47
48  # define TX_WATER (0)
49  # define TX_GRASS (1)
50  # define TX_DIRT (2)
51  # define TX_STONE (3)
52  # define TX_SNOW (4)
53  # define TX_MAX (5)
54
55  typedef struct  s_image {
56      int w, h;
57  }               t_image;
58
59  typedef struct  s_texture {
60      t_image * image;
```

```
61      GLuint  txID;
62  }               t_texture;
63
64  /** a terrain */
65  typedef struct  s_terrain {
66      t_vec2i index;
67      t_mat4f mat;
68      GLuint  vao;
69      GLuint  vbo;
70      float   * vertices;
71      int     initialized;
72  }               t_terrain;
73
74  # define WORLD_OCTAVES (10)
75
76  /** the world */
77  typedef struct  s_world {
78      t_hmap          * terrains;
79      t_noise         * octaves[WORLD_OCTAVES];
80      t_array_list    * generators;
81      float           max_height;
82      t_image         * heightmap;
83      int             time;
84  }               t_world;
85
86  typedef struct  s_generator {
87      float   (*heightGen)(t_world *, struct s_generator *, float, float);
88      int     (*colorGen)(t_world *, struct s_generator *, float, float, float);
89      int     (*canGenerateAt)(t_world *, struct s_generator *, float, float);
90      float   heightGenStep;
91      int     octaves;
92      float   amplitude;
93      float   persistance;
94      float   frequency;
95      float   lacunarity;
96  }               t_generator;
97
98  /** the renderer part of the program */
99  typedef struct  s_renderer {
100     t_glh_program   * program; //the rendering GPU program
101     GLuint          terrain_indices; //terrain indices buffer
102     GLuint          terrain_vertices; //terrain vertices buffer (static grid)
103     t_array_list    * render_list; //the list of terrain to render
104     t_array_list    * delete_list; //the list of terrain to delete
105     t_texture       texture;
106     int             vertexCount; //number of vertices drawn on last frame
107     t_vec3f         sunray; //sun light vector
108 }               t_renderer;
109
110 typedef struct  s_env {
111     t_glh_context   * context;
112     t_world         world;
113     t_renderer      renderer;
114     t_camera        camera; //user camera
115     thrd_t          thrd;
116     int             is_running;
117 }               t_env;
118
119 extern t_env g_env;
120
121 //get env
122 t_env * getEnv(void);
123
124 //renderer related functions
125 void rendererInit(t_renderer * renderer);
126 void rendererDelete(t_renderer * renderer);
127 void rendererUpdate(t_glh_context * context, t_world * world, t_renderer * renderer, t_camera *
        camera);
128 void rendererRender(t_glh_context * context, t_world * world, t_renderer * renderer, t_camera *
        camera);
129
130 //world related functions
131 void        worldInit(t_world * world, char * bmpfile, float maxheight, long seed);
132 void        worldDelete(t_world * world);
133 void        worldUpdate(t_glh_context * context, t_world * world, t_renderer * renderer, t_camera
         * camera);
134 void        worldGetGridIndex(t_world * world, float worldX, float worldZ, int * gridX, int *
        gridY);
135 t_terrain * worldGetTerrain(t_world * world, int gridX, int gridY);
136 t_generator *   worldGetGeneratorAt(t_world * world, float wx, float wz);
137
138 /** generators */
139 void    generatorsInit(t_world * world);
140 void    generatorsDelete(t_world * world);
141
142 //terrains
143 t_terrain * terrainNew(t_world * world, int gridX, int gridY);
```

```
144 void        terrainDelete(t_terrain * terrain);
145 int         terrainHash(t_terrain * terrain);
146 int         terrainCmp(t_terrain * left, t_terrain * right);
147 void        terrainLoadHeightMap(t_terrain * terrains, int * n, char const * bmpfile);
148 void        terrainGenerate(t_world * world, t_terrain * terrain);
149
150 //camera related functions
151 void cameraInit(t_camera * camera);
152 void cameraDelete(t_camera * camera);
153 void cameraUpdate(t_glh_context * context, t_world * world, t_renderer * renderer, t_camera *
        camera);
154
155 //heightmaps (bmp file)
156 t_image      *  imageNew(char const * path);
157 void            imageDelete(t_image * t_image);
158 int             heightmapGetHeight(t_image * image, float x, float y);
159
160 //inputs
161 void inputInit(t_glh_context * context);
162 void inputUpdate(t_glh_context * context, t_world * world, t_renderer * renderer, t_camera *
        camera);
163
164 #endif
```

renderer/srcs/camera.c

```
1  #include "renderer.h"
2
3  void cameraInit(t_camera * camera) {
4      camera->pos.x = TERRAIN_SIZE * 2, camera->pos.y = TERRAIN_SIZE * 2, camera->pos.z =
           TERRAIN_SIZE * 2;
5      camera->rot.pitch = 0, camera->rot.yaw = 0, camera->rot.roll = 0;
6      camera->fov = DEG_TO_RAD(70.0f);
7      camera->near_distance = 0.01f;
8      camera->far_distance = TERRAIN_RENDER_DISTANCE * TERRAIN_RENDER_DISTANCE * TERRAIN_SIZE;
9      camera->movespeed = 2.0f;
10 }
11
12 void cameraDelete(t_camera * camera) {
13 }
14
15 static void cameraUpdateMatrices(t_camera * camera) {
16
17     //matrices
18     t_vec3f * viewvec = &(camera->vview);
19     t_mat4f * view = &(camera->mview);
20     t_mat4f * proj = &(camera->mproj);
21     t_mat4f * viewproj = &(camera->mviewproj);
22
23     //view vector
24     float pitch = DEG_TO_RAD(camera->rot.pitch);
25     float yaw = DEG_TO_RAD(camera->rot.yaw);
26     float roll = DEG_TO_RAD(camera->rot.roll);
27     float cospitch = cos(pitch);
28     viewvec->x = cospitch * sin(yaw);
29     viewvec->y = -sin(pitch);
30     viewvec->z = -cospitch * cos(yaw);
31     vec3f_normalize(viewvec, viewvec);
32
33     //view matrix
34     mat4f_identity(view);
35     mat4f_rotateX(view, view, pitch);
36     mat4f_rotateY(view, view, yaw);
37     mat4f_rotateZ(view, view, roll);
38     mat4f_translate(view, view, -camera->pos.x, -camera->pos.y, -camera->pos.z);
39
40     //projection matrix
41     float aspect = 1.6f;
42     mat4f_perspective(proj, aspect, camera->fov, camera->near_distance, camera->far_distance);
43
44     //combine view and projection
45     mat4f_mult(viewproj, proj, view);
46 }
47
48 void cameraUpdate(t_glh_context * context, t_world * world, t_renderer * renderer, t_camera *
        camera) {
49     //update camera matrices
50     cameraUpdateMatrices(camera);
51     //update camera world index
52     worldGetGridIndex(world, camera->pos.x, camera->pos.z, &(camera->terrain_index.x), &(camera->
           terrain_index.y));
53 }
```

renderer/srcs/generator.c

```c
#include "renderer.h"

static float clamp(float val, float min, float max) {
    return (val > max ? max : val < min ? min : val);
}

static int mountainGenColor(t_world * world, t_generator * generator, float wx, float wy, float wz) {
    float r = wy / world->max_height;
    if (r <= 0.08f) {
        return (TX_WATER);
    } else if (r <= 0.60f) {
        return (TX_GRASS);
    } else if (r <= 0.75f) {
        return (TX_DIRT);
    } else if(r <= 0.90f) {
        return (TX_STONE);
    return (TX_SNOW);
}

static float normalizeHeight(t_world * world, float heightFactor) {
    heightFactor += 1;
    heightFactor *= 0.5f;
    heightFactor = clamp(heightFactor, 0.0f, 1.0f);
    return (world->max_height * heightFactor);
}

/** the height generator function for moutains */
static float genNoiseHeight(t_world * world, t_generator * generator, float wx, float wz) {
    float heightFactor = 0.0f;
    float frequency = generator->frequency;
    float amplitude = generator->amplitude;
    for (int i = 0 ; i < generator->octaves ; i++) {
        heightFactor += pnoise2(world->octaves[i], wx * frequency, wz * frequency) * amplitude;
        frequency *= generator->lacunarity;
        amplitude *= generator->persistance;
    }
    return (normalizeHeight(world, heightFactor));
}

static float heightmapGenHeight(t_world * world, t_generator * generator, float wx, float wz) {
    int px = (int)(wx / TERRAIN_UNIT);
    int py = (int)(wz / TERRAIN_UNIT);
    px = clamp(px, 0, world->heightmap->w - 1);
    py = clamp(py, 0, world->heightmap->h - 1);

    int rgb = heightmapGetHeight(world->heightmap, px, py);
    float height = rgb / (255.0f * 3.0f);
    return (height * world->max_height);
}

static int heightmapCanGenerate(t_world * world, t_generator * generator, float wx, float wz) {
    int px = (int)(wx / TERRAIN_UNIT);
    int py = (int)(wz / TERRAIN_UNIT);
    return (px >= 0 && py >= 0 && px < world->heightmap->w && py < world->heightmap->h);
}

static int canGenerate(t_world * world, t_generator * generator, float wx, float wz) {
    return (1);
}

static void registerGenerator(t_world * world,
                              float (*heightGen)(t_world *, struct s_generator *, float, float),
                              int (*colorGen)(t_world *, struct s_generator *, float, float, float),
                              int (*canGen)(t_world *, struct s_generator *, float, float),
                              float heightGenStep, int octaves,
                              float amplitude, float persistance,
                              float frequency, float lacunarity) {
    t_generator generator;
    generator.heightGen = heightGen;
    generator.colorGen = colorGen;
    generator.canGenerateAt = canGen;
    generator.heightGenStep = heightGenStep;
    generator.octaves = octaves;
    generator.amplitude = amplitude;
    generator.frequency = frequency;
    generator.persistance = persistance;
    generator.lacunarity = lacunarity;
    array_list_add(world->generators, &generator);
}

void generatorsInit(t_world * world) {
    if (world->heightmap == NULL) {
```

```
83          printf("No heightmaps set, generating terrain procedurally\n");
84          float step = TERRAIN_UNIT / 16.0f;
85          registerGenerator(world, genNoiseHeight, mountainGenColor, canGenerate, step, 4, 1.0f,
                0.5f, 0.03f, 2.0f);
86      } else {
87          printf("Heightmap in use\n");
88          registerGenerator(world, heightmapGenHeight, mountainGenColor, heightmapCanGenerate,
                TERRAIN_UNIT, 0, 0, 0, 0, 0);
89      }
90  }
91
92  void generatorsDelete(t_world * world) {
93      array_list_delete(world->generators);
94      free(world->generators);
95  }
```

renderer/srcs/heightmap.c

```
1   #include "renderer.h"
2
3   int heightmapGetHeight(t_image * image, float x, float y) {
4       int idx = ((int)x * image->h + (int)y) * 3;
5       unsigned char * rgb = (unsigned char*)(image + 1);
6       unsigned char b = rgb[idx + 0];
7       unsigned char g = rgb[idx + 1];
8       unsigned char r = rgb[idx + 2];
9       return (r + g + b);
10  }
```

renderer/srcs/image.c

```
1   #include "renderer.h"
2
3   # define BMP_HEADER_SIZE (54)
4
5   t_image * imageNew(char const * path) {
6
7       int fd = open(path, O_RDONLY);
8       if (fd == -1) {
9           return (NULL);
10      }
11
12      char header[BMP_HEADER_SIZE];
13      read(fd, &header, sizeof(header));
14
15      //magic
16      if (header[0] != 'B' || header[1] != 'M') {
17          close(fd);
18          return (NULL);
19      }
20
21      int offset = *((int*)(header + 0x0A));
22      int w = *((int*)(header + 0x12));
23      int h = *((int*)(header + 0x16));
24      t_image * image = (t_image*)malloc(sizeof(t_image) + 3 * w * h);
25      if (image == NULL) {
26          close(fd);
27          return (NULL);
28      }
29      image->w = w;
30      image->h = h;
31
32      //read useless bytes
33      lseek(fd, offset - BMP_HEADER_SIZE, SEEK_CUR);
34
35      //read raw bytes
36      read(fd, image + 1, w * h * 3);
37      close(fd);
38
39      return (image);
40  }
41
42  void imageDelete(t_image * map) {
43      free(map);
44  }
```

renderer/srcs/input.c

```c
#include "renderer.h"

static void inputKey(GLFWwindow * winptr, int key, int scancode, int action, int mods) {
    //close
    if (key == GLFW_KEY_ESCAPE) {
        glfwSetWindowShouldClose(winptr, 1);
    }
}

static void inputUpdateCamera(t_camera * camera) {

    float movespeed = camera->movespeed;
    static float rotspeed = 0.3f;

    t_glh_window * win = glhGetWindow();

    //camera speed
    if (glfwGetKey(win->pointer, GLFW_KEY_KP_ADD) == GLFW_PRESS) {
        camera->movespeed *= 1.2f;
    } else if (glfwGetKey(win->pointer, GLFW_KEY_KP_SUBTRACT) == GLFW_PRESS) {
        camera->movespeed *= 0.833f;
    }

    //rotation
    camera->rot.pitch += ((win->mouseY - win->prev_mouseY) * rotspeed);
    camera->rot.yaw += ((win->mouseX - win->prev_mouseX) * rotspeed);

    //move
    if (glfwGetKey(win->pointer, GLFW_KEY_W) == GLFW_PRESS) {
        camera->pos.x += camera->vview.x * movespeed;
        camera->pos.y += camera->vview.y * movespeed;
        camera->pos.z += camera->vview.z * movespeed;
    } else if (glfwGetKey(win->pointer, GLFW_KEY_S) == GLFW_PRESS) {
        camera->pos.x += -camera->vview.x * movespeed;
        camera->pos.y += -camera->vview.y * movespeed;
        camera->pos.z += -camera->vview.z * movespeed;
    }

    if (glfwGetKey(win->pointer, GLFW_KEY_D) == GLFW_PRESS) {
        camera->pos.x += -camera->vview.z * movespeed;
        camera->pos.z += camera->vview.x * movespeed;
    }
    else if (glfwGetKey(win->pointer, GLFW_KEY_A) == GLFW_PRESS) {
        camera->pos.x += camera->vview.z * movespeed;
        camera->pos.z += -camera->vview.x * movespeed;
    }
}

void inputUpdate(t_glh_context * context, t_world * world, t_renderer * renderer, t_camera *
    camera) {
    inputUpdateCamera(camera);
    inputUpdateDebug(context, world, renderer, camera);
}

void inputInit(t_glh_context * context) {
    glfwSetInputMode(context->window->pointer, GLFW_CURSOR, GLFW_CURSOR_DISABLED);
    glfwSetKeyCallback(context->window->pointer, inputKey);
}
```

renderer/srcs/main.c

```c
#include "renderer.h"

void printUsage(char * binary, FILE * dst) {
    fprintf(dst, "usage: ./%s [FLAGS]\n", binary);
    fprintf(dst, "flags available are:\n");
    fprintf(dst, "\t-r {SEED} for random/infinite terrain generation\n");
    fprintf(dst, "\t-f [FILE] for a bmp terrain heightmap loading\n");
    fprintf(dst, "\t-h [MAX_HEIGHT] to define maximum height of meshes\n");
    fprintf(dst, "examples:\n");
    fprintf(dst, "\t./%s -r 42 -h 256\n", binary);
    fprintf(dst, "\t./%s -t \"texture.bmp\" -h 12\n", binary);
}

t_env g_env;

t_env * getEnv(void) {
    return (&(g_env));
}

static int threadLoop(void * args) {
    t_env           * env       = getEnv();
    t_glh_context   * context   = env->context;
```

```
23      t_renderer      * renderer  = &(env->renderer);
24      t_world         * world     = &(env->world);
25      t_camera        * camera    = &(env->camera);
26
27      printf("Thread loop started!\n");
28
29      while (env->is_running) {
30          //update the camera and the world
31          worldUpdate(context, world, renderer, camera);
32
33          //10 ups
34          usleep(50 * 1000);
35
36      }
37
38      printf("Thread loop stopped!\n");
39
40      return (0);
41  }
42
43  int main(int argc, char **argv) {
44
45      //get binary name
46      char * binary = argc == 0 ? "renderer" : argv[0];
47
48      //parse arguments
49      int optind;
50      char mode = 'r';
51      long seed = time(NULL);
52      float maxheight = 1.0f;
53      char * bmpfile = NULL;
54      for (optind = 1; optind < argc; optind++) {
55
56          if (argv[optind][0] != '-' || argv[optind][2] != 0) {
57               printUsage(binary, stderr); return (EXIT_FAILURE);
58          }
59
60          mode = argv[optind][1];
61
62          switch (argv[optind][1]) {
63              case 'r': if (optind + 1 < argc) { seed = atoi(argv[++optind]); } break;
64              case 'f': if (optind + 1 >= argc) { printUsage(binary, stderr); return (EXIT_FAILURE)
                    ; } else { bmpfile = strdup(argv[++optind]); } break;
65              case 'h': if (optind + 1 >= argc) { printUsage(binary, stderr); return (EXIT_FAILURE)
                    ; } else { maxheight = atof(argv[++optind]); } break;
66              default: printUsage(binary, stderr); return (EXIT_FAILURE);
67          }
68      }
69
70      if (argc <= 1) {
71          printUsage(binary, stdout);
72          printf("\n");
73      }
74      printf("{mode='%c'}, {seed='%ld'}, {maxheight='%f'}, {bmpfile='%s'}\n\n", mode, seed,
            maxheight, bmpfile);
75
76      printf("Initializing openGL...\n");
77
78      glhInit();
79
80      printf("Creating gl context...\n");
81      t_env * env = getEnv();
82
83      env->context = glhCreateContext();
84
85      t_glh_context * context = env->context;
86
87      if (context == NULL) {
88          fprintf(stderr, "Failed to create gl context.\n");
89          return (EXIT_FAILURE);
90      }
91
92      if (context->window == NULL) {
93          fprintf(stderr, "Failed to create gl window.\n");
94          return (EXIT_FAILURE);
95      }
96
97      printf("Making gl context current...\n");
98      glhMakeContextCurrent(context);
99
100     t_world * world         = &(env->world);
101     t_renderer * renderer   = &(env->renderer);
102     t_camera * camera       = &(env->camera);
103     thrd_t * thrd           = &(env->thrd);
104
105     printf("Initializing camera...\n");
106     cameraInit(camera);
```

```c

    printf("Initializing renderer...\n");
    rendererInit(renderer);

    printf("Initializing inputs...\n");
    inputInit(context);

    printf("Initializing world...\n");
    worldInit(world, bmpfile, maxheight, seed);

    printf("Creating calculator thread...\n");
    thrd_create(thrd, threadLoop, &env);

    printf("Rendering started...\n");

    long int total = 0;
    long int count = 0;

    env->is_running = 1;

    long t1, t2;
    while (!glhWindowShouldClose(context->window) && env->is_running) {

        MICROSEC(t1);

        //update the window
        glhWindowUpdate(context->window);

        //input
        inputUpdate(context, world, renderer, camera);

        //camera
        cameraUpdate(context, world, renderer, camera);

        //update the renderer
        rendererUpdate(context, world, renderer, camera);

        //render
        rendererRender(context, world, renderer, camera);

        MICROSEC(t2);
        total += (t2 - t1);
        count++;

        //swap buffers
        glhSwapBuffer(context->window);
    }

    env->is_running = 0;

    //wait for calculator thread to finish
    printf("Waiting for thread to finish...\n");
    thrd_join(env->thrd, NULL);

    printf("Loop ended\n");

    printf("Deleting camera...\n");
    cameraDelete(camera);

    printf("Deleting world...\n");
    worldDelete(world);

    printf("Deleting renderer...\n");
    rendererDelete(renderer);

    printf("Destroying gl context..\n");
    glhDestroyContext(context);

    printf("Stopping openGL...\n");
    glhStop();

    printf("All done\n");

    printf("Moyenne: %ld\n", total / 1000 / count);

    return (0);
}
```

renderer/srcs/renderer.c

```c
#include "renderer.h"

GLuint u_mvp_matrix;
GLuint u_transf_matrix;
```

```
5  GLuint u_sky_color;
6  GLuint u_time;
7  GLuint u_sunpos;
8
9  static void rendererBindAttributes(t_glh_program * program) {
10     glhProgramBindAttribute(program, 0, "pos");
11     glhProgramBindAttribute(program, 1, "uv");
12     glhProgramBindAttribute(program, 2, "height");
13     glhProgramBindAttribute(program, 3, "normal");
14     glhProgramBindAttribute(program, 4, "textureID");
15 }
16
17 static void rendererLinkUniforms(t_glh_program * program) {
18     u_mvp_matrix = glhProgramGetUniform(program, "mvp_matrix");
19     u_transf_matrix = glhProgramGetUniform(program, "transf_matrix");
20     u_sky_color = glhProgramGetUniform(program, "sky_color");
21     u_time = glhProgramGetUniform(program, "time");
22     u_sunpos = glhProgramGetUniform(program, "sunray");
23 }
24
25 static void rendererGenerateBufferIndices(t_renderer * renderer) {
26
27     long size = sizeof(unsigned short) * (TERRAIN_DETAIL - 1) * (TERRAIN_DETAIL - 1) * 6;
28     unsigned short * indices = (unsigned short *) malloc(size);
29
30     int x, z;
31     int i00, i01, i11, i10;
32     int i = 0;
33     for (x = 0 ; x < TERRAIN_DETAIL - 1; x++) {
34         for (z = 0 ; z < TERRAIN_DETAIL - 1; z++) {
35
36             i00 = x * TERRAIN_DETAIL + z;
37             i01 = i00 + 1;
38             i10 = (x + 1) * TERRAIN_DETAIL + z;
39             i11 = i10 + 1;
40             indices[i++] = i00;
41             indices[i++] = i11;
42             indices[i++] = i10;
43             indices[i++] = i00;
44             indices[i++] = i01;
45             indices[i++] = i11;
46         }
47     }
48
49     renderer->terrain_indices = glhVBOGen();
50     glhVBOBind(GL_ELEMENT_ARRAY_BUFFER, renderer->terrain_indices);
51     glhVBOData(GL_ELEMENT_ARRAY_BUFFER, size, indices, GL_STATIC_DRAW);
52     glhVBOUnbind(GL_ELEMENT_ARRAY_BUFFER);
53
54     free(indices);
55 }
56
57 static void rendererGenerateBufferVertices(t_renderer * renderer) {
58
59     long size = sizeof(float) * 4 * TERRAIN_DETAIL * TERRAIN_DETAIL;
60     float * vertices = (float *) malloc(size);
61
62     float unit = 1 / (float)(TERRAIN_DETAIL - 1);
63     int x, z;
64     int i = 0;
65     for (x = 0 ; x < TERRAIN_DETAIL ; x++) {
66         for (z = 0 ; z < TERRAIN_DETAIL ; z++) {
67             vertices[i++] = x * unit;
68             vertices[i++] = z * unit;
69             vertices[i++] = x % 2 == 0 ? 0.0f : 1.0f;
70             vertices[i++] = z % 2 == 0 ? 0.0f : 1.0f;
71         }
72     }
73
74     renderer->terrain_vertices = glhVBOGen();
75     glhVBOBind(GL_ARRAY_BUFFER, renderer->terrain_vertices);
76     glhVBOData(GL_ARRAY_BUFFER, size, vertices, GL_STATIC_DRAW);
77     glhVBOUnbind(GL_ARRAY_BUFFER);
78
79     free(vertices);
80 }
81
82 static void rendererGenerateBuffers(t_renderer * renderer) {
83     rendererGenerateBufferIndices(renderer);
84     rendererGenerateBufferVertices(renderer);
85 }
86
87 void rendererInit(t_renderer * renderer) {
88
89     //init math lib
90     cmaths_init();
91
```

```
92      //create the program
93      renderer->program = glhProgramNew();
94
95      //load shaders
96      GLuint fs = glhShaderLoad("./shaders/terrain.fs", GL_FRAGMENT_SHADER);
97      GLuint vs = glhShaderLoad("./shaders/terrain.vs", GL_VERTEX_SHADER);
98      glhProgramAddShader(renderer->program, fs, GLH_SHADER_FRAGMENT);
99      glhProgramAddShader(renderer->program, vs, GLH_SHADER_VERTEX);
100
101     //link
102     glhProgramLink(renderer->program, rendererBindAttributes, rendererLinkUniforms);
103
104     //generate terrain indices
105     rendererGenerateBuffers(renderer);
106
107     //initialize lists
108     renderer->render_list = array_list_new(256, sizeof(t_terrain *));
109     renderer->delete_list = array_list_new(256, sizeof(t_terrain *));
110
111     //image
112     renderer->texture.txID = glhGenTexture();
113     renderer->texture.image = imageNew("./res/textures.bmp");
114     unsigned char * pixels = (unsigned char*)(renderer->texture.image + 1);
115     glBindTexture(GL_TEXTURE_2D, renderer->texture.txID);
116     printf("txID: %u w : %d h : %d\n", renderer->texture.txID, renderer->texture.image->w,
            renderer->texture.image->h);
117     glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, renderer->texture.image->w, renderer->texture.image->h
            , 0, GL_BGR, GL_UNSIGNED_BYTE, pixels);
118
119     glGenerateMipmap(GL_TEXTURE_2D);
120     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR);
121     glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_LOD_BIAS, -1.6f);
122
123     //enable depth test
124     glEnable(GL_DEPTH_TEST);
125     glEnable(GL_CULL_FACE);
126     glCullFace(GL_BACK);
127
128     //set default sun ray
129     vec3f_set(&(renderer->sunray), 1.0f, 1.0f, 1.0f);
130     vec3f_normalize(&(renderer->sunray), &(renderer->sunray));
131
132     //swap interval for 60 fps max
133     glfwSwapInterval(1);
134 }
135
136 static void rendererInitTerrain(t_renderer * renderer, t_terrain * terrain) {
137
138     terrain->initialized = 1;
139
140     //allocate terrain model on GPU
141     terrain->vao = glhVAOGen();
142     terrain->vbo = glhVBOGen();
143
144     //bind vao
145     glhVAOBind(terrain->vao);
146
147     //bind indices
148     glhVBOBind(GL_ELEMENT_ARRAY_BUFFER, renderer->terrain_indices);
149
150     //bind static grid
151     glhVBOBind(GL_ARRAY_BUFFER, renderer->terrain_vertices);
152     glhVAOSetAttribute(0, 2, GL_FLOAT, 0, 4 * sizeof(float), NULL); //default vertices pos
153     glhVAOSetAttribute(1, 2, GL_FLOAT, 0, 4 * sizeof(float), (void*)(2 * sizeof(float))); //
            default vertices uv
154     glhVBOUnbind(GL_ARRAY_BUFFER);
155     glhVAOEnableAttribute(0);
156     glhVAOEnableAttribute(1);
157
158     //bind buffer
159     glhVBOBind(GL_ARRAY_BUFFER, terrain->vbo);
160     //set attruibutes
161     glhVAOSetAttribute(2, 1, GL_FLOAT, 0, TERRAIN_VERTEX_SIZE, NULL); //height
162     glhVAOSetAttribute(3, 2, GL_FLOAT, 0, TERRAIN_VERTEX_SIZE, (void*)(1 * sizeof(float))); //
            normal
163     glhVAOSetAttributeI(4, 3, GL_INT, TERRAIN_VERTEX_SIZE, (void*)((2 + 1) * sizeof(float))); //
            texture ID
164     glhVBOUnbind(GL_ARRAY_BUFFER);
165     //enable attributes
166     glhVAOEnableAttribute(2);
167     glhVAOEnableAttribute(3);
168     glhVAOEnableAttribute(4);
169
170     //unbind vao
171     glhVAOUnbind();
172 }
173
```

```
174 void rendererUpdate(t_glh_context * context, t_world * world, t_renderer * renderer, t_camera *
        camera) {
175
176     //clear lists
177     array_list_clear(renderer->render_list);
178     array_list_clear(renderer->delete_list);
179
180     //update lists
181     HMAP_ITER_START(world->terrains, t_terrain *, terrain) {
182
183         t_vec3f diff;
184         diff.x = terrain->index.x - camera->terrain_index.x;
185         diff.y = 0;
186         diff.z = terrain->index.y - camera->terrain_index.y;
187
188             //if to far, delete this terrain
189         if (diff.x <= -TERRAIN_KEEP_LOADED_DISTANCE || diff.z <= -TERRAIN_KEEP_LOADED_DISTANCE ||
190             diff.x >= TERRAIN_KEEP_LOADED_DISTANCE || diff.z >= TERRAIN_KEEP_LOADED_DISTANCE) {
191             array_list_add(renderer->delete_list, &terrain);
192         } else {
193
194             float distance = vec3f_length(&diff);
195             if (distance < TERRAIN_RENDER_DISTANCE) {
196                 float normalizer = 1 / distance;
197                 diff.x *= normalizer;
198                 diff.z *= normalizer;
199
200                 float dot = vec3f_dot_product(&(camera->vview), &diff);
201                 if (distance <= 2 || acos_f(dot) < camera->fov) {
202                     array_list_add(renderer->render_list, &terrain);
203                 }
204             }
205         }
206     }
207     HMAP_ITER_END(world->terrains, t_terrain *, terrain);
208
209     //remove terrains
210     ARRAY_LIST_ITER_START(renderer->delete_list, t_terrain **, terrain_ptr, i) {
211         t_terrain * terrain = *terrain_ptr;
212         hmap_remove_key(world->terrains, &(terrain->index));
213         terrainDelete(terrain);
214     }
215     ARRAY_LIST_ITER_END(renderer->delete_list, t_terrain **, terrain_ptr, i);
216
217     array_list_clear(renderer->delete_list);
218 }
219
220 static int rendererRenderTerrain(t_renderer * renderer, t_terrain * terrain) {
221     static int vertexCount = (TERRAIN_DETAIL - 1) * (TERRAIN_DETAIL - 1) * 6;
222
223     //if it vertices arent up to date
224     if (terrain->vertices != NULL) {
225         //update them
226         glhVBOBind(GL_ARRAY_BUFFER, terrain->vbo);
227         glhVBOData(GL_ARRAY_BUFFER, TERRAIN_DETAIL * TERRAIN_DETAIL * TERRAIN_VERTEX_SIZE,
                terrain->vertices, GL_STATIC_DRAW);
228         //release data
229         free(terrain->vertices);
230         terrain->vertices = NULL;
231         glhVBOUnbind(GL_ARRAY_BUFFER);
232     }
233
234     //load the matrix as a uniform variable
235     glhProgramLoadUniformMatrix4f(u_transf_matrix, (float*)(&(terrain->mat)));
236
237     //sun light
238     glhProgramLoadUniformVec3f(u_sunpos, renderer->sunray.x, renderer->sunray.y, renderer->sunray
            .z);
239
240     //bind the model
241     glhVAOBind(terrain->vao);
242
243     //draw it
244     glhDrawElements(GL_TRIANGLES, vertexCount, GL_UNSIGNED_SHORT, NULL);
245
246     return (vertexCount);
247 }
248
249 static void rendererPrepareProgram(t_glh_context * context, t_world * world, t_renderer *
        renderer, t_camera * camera) {
250     //set the texture
251     glActiveTexture(GL_TEXTURE0);
252     glBindTexture(GL_TEXTURE_2D, renderer->texture.txID);
253
254     //bind the program
255     glhProgramUse(renderer->program);
256
```

```
257     //load uniforms
258     glhProgramLoadUniformMatrix4f(u_mvp_matrix, (float*)&(camera->mviewproj));
259     glhProgramLoadUniformVec3f(u_sky_color, 0.46f, 0.70f, 0.99f);
260     glhProgramLoadUniformInt(u_time, world->time);
261 }
262
263 void rendererRender(t_glh_context * context, t_world * world, t_renderer * renderer, t_camera *
        camera) {
264     //viewport
265     glhViewPort(0, 0, context->window->width, context->window->height);
266
267     //clear color buffer
268     glhClearColor(0.46f, 0.70f, 0.99f, 1.0f);
269     glhClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
270
271     //prepare the renderer (bind program, textures and uniforms)
272     rendererPrepareProgram(context, world, renderer, camera);
273
274     //total vertex drawn
275     renderer->vertexCount = 0;
276
277     //for every terrain which has to be rendered
278     ARRAY_LIST_ITER_START(renderer->render_list, t_terrain **, terrain_ptr, i) {
279
280         //get the terrain
281         t_terrain * terrain = *terrain_ptr;
282
283         //if it is not initialized, initialize it
284         if (!terrain->initialized) {
285             rendererInitTerrain(renderer, terrain);
286         }
287
288         if (terrain->initialized) {
289             renderer->vertexCount += rendererRenderTerrain(renderer, terrain);
290         }
291     }
292     ARRAY_LIST_ITER_END(renderer->render_list, t_terrain *, terrain, i);
293
294     glhVAOUnbind();
295     glhProgramUse(NULL);
296 }
297
298
299 void rendererDelete(t_renderer * renderer) {
300     cmaths_deinit();
301     glhProgramDelete(renderer->program);
302     glhVBODelete(renderer->terrain_indices);
303     glhVBODelete(renderer->terrain_vertices);
304     array_list_delete(renderer->render_list);
305     free(renderer->render_list);
306     imageDelete(renderer->texture.image);
307     glhDeleteTexture(renderer->texture.txID);
308 }
```

renderer/srcs/terrain.c

```
1  #include "renderer.h"
2
3  static void terrainCalculateNormal(t_world * world, t_generator * generator, float * nx, float *
       nz,
4                                     float wx, float wy, float wz) {
5      float dx = generator->heightGenStep;
6      float dz = generator->heightGenStep;
7      *nx = (generator->heightGen(world, generator, wx + dx, wz) - wy) / dx;
8      *nz = (generator->heightGen(world, generator, wx, wz + dz) - wy) / dz;
9  }
10
11 static void terrainGenerateVertices(t_world * world, float vertices[], int gridX, int gridY) {
12     float nx, nz;
13     int i = 0;
14     for (int x = 0 ; x < TERRAIN_DETAIL ; x++) {
15         for (int y = 0 ; y < TERRAIN_DETAIL; y++) {
16
17             float wx = (gridX * (TERRAIN_DETAIL - 1) + x) * TERRAIN_UNIT;
18             float wz = (gridY * (TERRAIN_DETAIL - 1) + y) * TERRAIN_UNIT;
19             t_generator * generator = worldGetGenerator(world, wx, wz);
20             float wy = generator->heightGen(world, generator, wx, wz);
21             int textureID = generator->colorGen(world, generator, wx, wy, wz);
22
23             terrainCalculateNormal(world, generator, &nx, &nz, wx, wy, wz);
24
25             *(vertices + i++) = wy;
26             *(vertices + i++) = nx;
27             *(vertices + i++) = nz;
```

```
            *((int*)(vertices + i++)) = textureID;
        }
    }
}

void terrainGenerate(t_world * world, t_terrain * terrain) {
    int gridX = terrain->index.x;
    int gridY = terrain->index.y;
    terrain->vertices = (float *) malloc(TERRAIN_DETAIL * TERRAIN_DETAIL * TERRAIN_VERTEX_SIZE);
    terrainGenerateVertices(world, terrain->vertices, gridX, gridY);
}

/** delete the given terrain */
void terrainDelete(t_terrain * terrain) {
    if (terrain->initialized) {
        terrain->initialized = 0;
        glhVAODelete(terrain->vao);
        glhVBODelete(terrain->vbo);
    }

    if (terrain->vertices != NULL) {
        free(terrain->vertices);
        terrain->vertices = NULL;
    }
    free(terrain);
}

/** allocate a new terrain on heap + gpu */
t_terrain * terrainNew(t_world * world, int gridX, int gridY) {

    //allocate the terrain
    t_terrain * terrain = (t_terrain*)malloc(sizeof(t_terrain));
    if (terrain == NULL) {
        return (NULL);
    }

    terrain->index.x = gridX;
    terrain->index.y = gridY;
    terrain->vertices = NULL;
    terrain->initialized = 0;

    //generate the transformation matrix for this terrain
    mat4f_identity(&(terrain->mat));
    mat4f_translate(&(terrain->mat), &(terrain->mat), terrain->index.x * TERRAIN_SIZE, 0, terrain
        ->index.y * TERRAIN_SIZE);
    mat4f_scale(&(terrain->mat), &(terrain->mat), TERRAIN_SIZE);

    terrainGenerate(world, terrain);

    return (terrain);
}
```

renderer/srcs/world.c

```
#include "renderer.h"

static void worldLoadHeightmap(t_world * world, char * file) {
    if (file == NULL) {
        world->heightmap = NULL;
    } else {
        world->heightmap = imageNew(file);
    }
}

static unsigned int world_vec2i_hash(t_vec2i * vec) {
    unsigned int hash = vec->x * (TERRAIN_KEEP_LOADED_DISTANCE * 2 + 1) + vec->y;
    return (hash);
}

void worldInit(t_world * world, char * bmpfile, float max_height, long seed) {
    glhCheckError("pre worldInit()");

    world->time = 0;
    world->max_height = max_height;
    worldLoadHeightmap(world, bmpfile);
    world->generators = array_list_new(16, sizeof(generator));
    generatorsInit(world);

    //create the terrain hash map
    world->terrains = hmap_new(TERRAIN_KEEP_LOADED_DISTANCE * 4, (t_hf)world_vec2i_hash, (t_cmpf)
        vec2i_nequals, (t_f)NULL, (t_f)NULL);
    if (world->terrains == NULL) {
        fprintf(stderr, "world.c : 1.10 : worldInit() : not enough memory\n");
        return ;
```

```
30      }
31
32      //noise creation
33      int i;
34      for (i = 0 ; i < WORLD_OCTAVES ; i++) {
35          world->octaves[i] = noiseNew();
36          noiseNextInt(&seed);
37          noiseSeed(world->octaves[i], seed);
38      }
39
40      glhCheckError("post worldInit()");
41  }
42
43  void worldDelete(t_world * world) {
44      if (world->heightmap != NULL) {
45          imageDelete(world->heightmap);
46      }
47
48      HMAP_ITER_START(world->terrains, t_terrain *, terrain) {
49          terrainDelete(terrain);
50      }
51      HMAP_ITER_END(world->terrains, t_terrain *, terrain);
52
53      hmap_delete(world->terrains);
54      free(world->terrains);
55
56      int i;
57      for (i = 0 ; i < WORLD_OCTAVES ; i++) {
58          noiseDelete(world->octaves[i]);
59      }
60
61      generatorsDelete(world);
62  }
63
64  t_generator * worldGetGeneratorAt(t_world * world, float wx, float wz) {
65      return (array_list_get(world->generators, 0));
66  }
67
68  void worldGetGridIndex(t_world * world, float worldX, float worldZ, int * gridX, int * gridY) {
69      *gridX = (int)worldX / TERRAIN_SIZE;
70      *gridY = (int)worldZ / TERRAIN_SIZE;
71
72      if (worldX < 0) {
73          *gridX -= 1;
74      }
75
76      if (worldZ < 0) {
77          *gridY -= 1;
78      }
79  }
80
81  t_terrain * worldGetTerrain(t_world * world, int gridX, int gridY) {
82      t_vec2i index;
83      index.x = gridX;
84      index.y = gridY;
85      return (hmap_get(world->terrains, &index));
86  }
87
88  static void worldLoadNewTerrains(t_world * world, t_camera * camera) {
89
90      int indexx = MAX(0, camera->terrain_index.x - TERRAIN_LOADED_DISTANCE);
91      int indexy = MAX(0, camera->terrain_index.y - TERRAIN_LOADED_DISTANCE);
92      int maxx = camera->terrain_index.x + TERRAIN_LOADED_DISTANCE;
93      int maxy = camera->terrain_index.y + TERRAIN_LOADED_DISTANCE;
94      int gridX, gridY;
95      for (gridX = indexx ; gridX  < maxx; gridX++) {
96          for (gridY = indexy ; gridY < maxy; gridY++) {
97
98              if (gridX < 0 || gridY < 0) {
99                  continue ;
100             }
101
102             //if this terrain isnt generated yet
103             if (worldGetTerrain(world, gridX, gridY) == NULL) {
104
105                 //can it be generated?
106                 float wx = gridX * TERRAIN_SIZE;
107                 float wz = gridY * TERRAIN_SIZE;
108                 t_generator * generator = worldGetGeneratorAt(world, wx, wz);
109                 if (!generator->canGenerateAt(world, generator, wx, wz)) {
110                     continue ;
111                 }
112
113                 //if so, generate it
114                 t_terrain * terrain = terrainNew(world, gridX, gridY);
115
116                 if (terrain != NULL) {
```

```
                    hmap_insert(world->terrains, terrain, &(terrain->index));
                }
            }
        }
    }
}

void worldUpdate(t_glh_context * context, t_world * world, t_renderer * renderer, t_camera *
    camera) {
    //load new terrains
    worldLoadNewTerrains(world, camera);
    world->time++;
}
```

2 Bibliothèque mathématiques

maths/includes/cmaths.h

```
#ifndef CMATHS_H
# define CMATHS_H

# include <math.h>
# include <stdlib.h>
# include <stdio.h>
# include <string.h>

# ifndef DEG_TO_RAD
#  define DEG_TO_RAD(X) (X * 0.01745329251f)
# endif

# ifndef RAD_TO_DEG
#  define RAD_TO_DEG(X) (X * 57.2957795131f)
# endif

# ifndef MAX
#  define MAX(X, Y) (X > Y ? X : Y)
# endif

# ifndef MIN
#  define MIN(X, Y) (X < Y ? X : Y)
# endif

# ifndef ABS
#  define ABS(X) (X < 0 ? -X : X)
# endif

int cmaths_init(void);
int cmaths_deinit(void);

float   acos_f(float x);
float   asin_f(float x);
float   atan_f(float x);

float   sin_f(float x);
float   cos_f(float x);
float   tan_f(float x);

float   sqrt_f(float x);

#endif
```

maths/includes/mat4f.h

```
#ifndef MAT4F_H
# define MAT4F_H

# include "cmaths.h"
# include "vec.h"

typedef struct  s_mat4f {
    float m00;
    float m01;
    float m02;
    float m03;

    float m10;
    float m11;
    float m12;
    float m13;
```

```
17
18      float m20;
19      float m21;
20      float m22;
21      float m23;
22
23      float m30;
24      float m31;
25      float m32;
26      float m33;
27  }               t_mat4f;
28
29  /** create a new matrix */
30  t_mat4f * mat4f_new(void);
31
32  /** delete the given matrix */
33  void mat4f_delete(t_mat4f * mat);
34
35  /** copy */
36  t_mat4f * mat4f_copy(t_mat4f * dst, t_mat4f * src);
37
38  /** set identity */
39  t_mat4f * mat4f_identity(t_mat4f * dst);
40
41  /** set to zero */
42  t_mat4f * mat4f_zero(t_mat4f * dst);
43
44  /** transpose */
45  t_mat4f * mat4f_transpose(t_mat4f * dst, t_mat4f * src);
46
47  /** scale */
48  t_mat4f * mat4f_scale(t_mat4f * dst, t_mat4f * mat, float f);
49
50  /** translate */
51  t_mat4f * mat4f_translate(t_mat4f * dst, t_mat4f * src, float tx, float ty, float tz);
52  t_mat4f * mat4f_translate3(t_mat4f * dst, t_mat4f * src, t_vec3f * translate);
53
54  /** rotate */
55  t_mat4f * mat4f_rotate(t_mat4f * dst, t_mat4f * src, float angle, t_vec3f * axis);
56  t_mat4f * mat4f_rotateX(t_mat4f * dst, t_mat4f * src, float angle);
57  t_mat4f * mat4f_rotateY(t_mat4f * dst, t_mat4f * src, float angle);
58  t_mat4f * mat4f_rotateZ(t_mat4f * dst, t_mat4f * src, float angle);
59  t_mat4f * mat4f_rotateXYZ(t_mat4f * dst, t_mat4f * src, t_vec3f * rot);
60
61  /** transformation matrix */
62  t_mat4f * mat4f_transformation(t_mat4f * dst, t_vec3f * translate, t_vec3f * rot, t_vec3f * scale
        );
63
64  /** determinant */
65  float mat4f_determinant(t_mat4f * mat);
66
67  /** invert */
68  t_mat4f * mat4f_invert(t_mat4f * dst, t_mat4f * src);
69
70  /** mult */
71  t_mat4f * mat4f_mult(t_mat4f * dst, t_mat4f * left, t_mat4f * right);
72
73  /** transform vec4f */
74  t_vec4f * mat4f_transform_vec4f(t_vec4f * dst, t_mat4f * left, t_vec4f * right);
75
76  /** projections matrix bellow: */
77
78  /** orthographic matrix */
79  t_mat4f * mat4f_orthographic(t_mat4f * dst, float left, float right, float bot, float top, float
        near, float far);
80
81  /** perspective matrix */
82  t_mat4f * mat4f_perspective(t_mat4f * dst, float aspect, float fov, float near, float far);
83
84  /** to string: return a string allocated with malloc() */
85  char * mat4f_str(t_mat4f * mat);
86
87  #endif
```

maths/includes/vec2i.h

```
1   #ifndef VEC2I_H
2   # define VEC2I_H
3
4   # include "cmaths.h"
5
6   typedef struct  s_vec2i {
7       union {
8           int x;
```

```
        int uvx;
    };

    union {
        int y;
        int uvy;
    };
}               t_vec2i;

/** create a new vec2i */
t_vec2i * vec2i_new(void);

/** delete the vec2i */
void vec2i_delete(t_vec2i * vec);

/** set the vec2i to 0 */
t_vec2i * vec2i_zero(t_vec2i * dst);

/** set the vec2i values */
t_vec2i * vec2i_set(t_vec2i * dst, int x, int y);
t_vec2i * vec2i_set2(t_vec2i * dst, t_vec2i * vec);

/** add two vec2i */
t_vec2i * vec2i_add(t_vec2i * dst, t_vec2i * left, t_vec2i * right);

/** sub two vec2i */
t_vec2i * vec2i_sub(t_vec2i * dst, t_vec2i * left, t_vec2i * right);

/** mult the vec2i by the given scalar */
t_vec2i * vec2i_mult(t_vec2i * dst, t_vec2i * vec, int scalar);
t_vec2i * vec2i_mult2(t_vec2i * dst, t_vec2i * left, t_vec2i * right);

/** scale product */
int vec2i_dot_product(t_vec2i * left, t_vec2i * right);

/** length */
int vec2i_length_squared(t_vec2i * vec);
int vec2i_length(t_vec2i * vec);

/** normalize */
t_vec2i * vec2i_normalize(t_vec2i * dst, t_vec2i * vec);

/** negate */
t_vec2i * vec2i_negate(t_vec2i * dst, t_vec2i * src);

/** angle between two vec */
int vec2i_angle(t_vec2i * left, t_vec2i * right);

/** mix the two vectors */
t_vec2i * vec2i_mix(t_vec2i * dst, t_vec2i * left, t_vec2i * right, int ratio);

/** comparison */
int vec2i_equals(t_vec2i * left, t_vec2i * right);

/** to string: return a string allocated with malloc() */
char * vec2i_str(t_vec2i * vec);

#endif
```

maths/includes/vec3f.h

```
#ifndef VEC3F_H
# define VEC3F_H

# include "cmaths.h"

typedef struct  s_vec3f {
    union {
        float x;
        float r;
        float pitch;
    };

    union {
        float y;
        float g;
        float yaw;
    };

    union {
        float z;
        float b;
        float roll;
    };
```

```
} t_vec3f;

/** create a new vec3f */
t_vec3f * vec3f_new(void);

/** delete the vec3f */
void vec3f_delete(t_vec3f * vec);

/** set the vec3f to 0 */
t_vec3f * vec3f_zero(t_vec3f * dst);

/** set the vec3f values */
t_vec3f * vec3f_set(t_vec3f * dst, float x, float y, float z);
t_vec3f * vec3f_set3(t_vec3f * dst, t_vec3f * vec);

/** add two vec3f */
t_vec3f * vec3f_add(t_vec3f * dst, t_vec3f * left, t_vec3f * right);

/** sub two vec3f */
t_vec3f * vec3f_sub(t_vec3f * dst, t_vec3f * left, t_vec3f * right);

/** mult the vec3f by the given scalar */
t_vec3f * vec3f_mult(t_vec3f * dst, t_vec3f * vec, float scalar);
t_vec3f * vec3f_mult3(t_vec3f * dst, t_vec3f * left, t_vec3f * right);

/** cross product */
t_vec3f * vec3f_cross(t_vec3f * dst, t_vec3f * left, t_vec3f * right);

/** scale product */
float vec3f_dot_product(t_vec3f * left, t_vec3f * right);

/** length */
float vec3f_length_squared(t_vec3f * vec);
float vec3f_length(t_vec3f * vec);

/** normalize */
t_vec3f * vec3f_normalize(t_vec3f * dst, t_vec3f * vec);

/** negate */
t_vec3f * vec3f_negate(t_vec3f * dst, t_vec3f * src);

/** angle between two vec */
float vec3f_angle(t_vec3f * left, t_vec3f * right);

/** mix the two vectors */
t_vec3f * vec3f_mix(t_vec3f * dst, t_vec3f * left, t_vec3f * right, float ratio);

/** comparison */
int vec3f_equals(t_vec3f * left, t_vec3f * right);

/** hash */
int vec3f_hash(t_vec3f * vec);

/** round vec3f */
t_vec3f * vec3f_round(t_vec3f * dst, t_vec3f * vec, int decimals);

/** to string: return a string allocated with malloc() */
char * vec3f_str(t_vec3f * vec);

#endif
```

maths/srcs/cmaths.c

```
#include "cmaths.h"

float * sin_table;

float atan_f(float x) {
    float xabs = ABS(x);
    return (0.78539816339f * x - x * (xabs - 1) * (0.2447f + 0.0663f * xabs));
}

float acos_f(float x) {
    //lagrange interpolation:
    return ((-0.69813170079773212f * x * x - 0.87266462599716477f) * x + 1.5707963267948966f);
}

float asin_f(float x) {
    return (-acos_f(x) + 1.5707963267948966f);
}

float tan_f(float x) {
    return (sin_f(x) / cos_f(x));
}
```

```
22
23  float sin_f(float x) {
24      unsigned int index = (unsigned int)RAD_TO_DEG(ABS(x)) % 360;
25      return (index > 180 ? -sin_table[index - 180] : sin_table[index]);
26  }
27
28  float cos_f(float x) {
29      return (sin_f(x + 1.5707963267948966f));
30  }
31
32  float sqrt_f(float x) {
33      unsigned int i = *(unsigned int*) &x;
34      i  += 127 << 23;
35      i >>= 1;
36      return (*(float*) &i);
37  }
38
39  int cmaths_init(void) {
40
41      sin_table = (float *) malloc(sizeof(float) * 4 * 360);
42      if (sin_table == NULL) {
43          return (0);
44      }
45
46      int i, j;
47      for (i = 0, j = 0 ; i < 180; i++) {
48          sin_table[j++] = (float)sin((double)DEG_TO_RAD((float)i));
49      }
50
51      return (1);
52  }
53
54  int cmaths_deinit(void) {
55      free(sin_table);
56      sin_table = NULL;
57      return (1);
58  }
```

maths/srcs/mat4f.c

```
1   #include "mat4f.h"
2
3   t_mat4f * mat4f_new(void) {
4       return ((t_mat4f*)malloc(sizeof(t_mat4f)));
5   }
6
7   void mat4f_delete(t_mat4f * mat) {
8       free(mat);
9   }
10
11  t_mat4f * mat4f_copy(t_mat4f * dst, t_mat4f * src) {
12      if (dst == NULL) {
13          if ((dst = mat4f_new()) == NULL) {
14              return (NULL);
15          }
16      }
17      memcpy(dst, src, sizeof(t_mat4f));
18      return (dst);
19  }
20
21  t_mat4f * mat4f_identity(t_mat4f * dst) {
22      if (dst == NULL) {
23          if ((dst = mat4f_new()) == NULL) {
24              return (NULL);
25          }
26      }
27
28      dst->m00 = 1, dst->m01 = 0, dst->m02 = 0, dst->m03 = 0;
29      dst->m10 = 0, dst->m11 = 1, dst->m12 = 0, dst->m13 = 0;
30      dst->m20 = 0, dst->m21 = 0, dst->m22 = 1, dst->m23 = 0;
31      dst->m30 = 0, dst->m31 = 0, dst->m32 = 0, dst->m33 = 1;
32      return (dst);
33  }
34
35  t_mat4f * mat4f_zero(t_mat4f * dst) {
36      if (dst == NULL) {
37          if ((dst = mat4f_new()) == NULL) {
38              return (NULL);
39          }
40      }
41      memset(dst, 0, sizeof(t_mat4f));
42      return (dst);
43  }
44
```

```
45  t_mat4f * mat4f_transpose(t_mat4f * dst, t_mat4f * src) {
46
47      if (dst == NULL) {
48          if ((dst = mat4f_new()) == NULL) {
49              return (NULL);
50          }
51      }
52
53      float m00 = src->m00;
54      float m01 = src->m10;
55      float m02 = src->m20;
56      float m03 = src->m30;
57      float m10 = src->m01;
58      float m11 = src->m11;
59      float m12 = src->m21;
60      float m13 = src->m31;
61      float m20 = src->m02;
62      float m21 = src->m12;
63      float m22 = src->m22;
64      float m23 = src->m32;
65      float m30 = src->m03;
66      float m31 = src->m13;
67      float m32 = src->m23;
68      float m33 = src->m33;
69
70      dst->m00 = m00, dst->m01 = m01, dst->m02 = m02, dst->m03 = m03;
71      dst->m10 = m10, dst->m11 = m11, dst->m12 = m12, dst->m13 = m13;
72      dst->m20 = m20, dst->m21 = m21, dst->m22 = m22, dst->m23 = m23;
73      dst->m30 = m30, dst->m31 = m31, dst->m32 = m32, dst->m33 = m33;
74
75      return (dst);
76  }
77
78  t_mat4f * mat4f_scale(t_mat4f * dst, t_mat4f * src, float scale) {
79
80      if (dst == NULL) {
81          if ((dst = mat4f_new()) == NULL) {
82              return (NULL);
83          }
84      }
85
86      dst->m00 = src->m00 * scale, dst->m01 = src->m01 * scale, dst->m02 = src->m02 * scale, dst->
            m03 = src->m03 * scale;
87      dst->m10 = src->m10 * scale, dst->m11 = src->m11 * scale, dst->m12 = src->m12 * scale, dst->
            m13 = src->m13 * scale;
88      dst->m20 = src->m20 * scale, dst->m21 = src->m21 * scale, dst->m22 = src->m22 * scale, dst->
            m23 = src->m23 * scale;
89
90      return (dst);
91  }
92
93  t_mat4f * mat4f_scale3(t_mat4f * dst, t_mat4f * src, t_vec3f * scale) {
94
95      if (dst == NULL) {
96          if ((dst = mat4f_new()) == NULL) {
97              return (NULL);
98          }
99      }
100
101     dst->m00 = src->m00 * scale->x, dst->m01 = src->m01 * scale->x, dst->m02 = src->m02 * scale->
            x, dst->m03 = src->m03 * scale->x;
102     dst->m10 = src->m10 * scale->y, dst->m11 = src->m11 * scale->y, dst->m12 = src->m12 * scale->
            y, dst->m13 = src->m13 * scale->y;
103     dst->m20 = src->m20 * scale->z, dst->m21 = src->m21 * scale->z, dst->m22 = src->m22 * scale->
            z, dst->m23 = src->m23 * scale->z;
104
105     return (dst);
106 }
107
108 t_mat4f * mat4f_translate(t_mat4f * dst, t_mat4f * src, float tx, float ty, float tz) {
109     if (dst == NULL) {
110         if ((dst = mat4f_new()) == NULL) {
111             return (NULL);
112         }
113     }
114
115     dst->m30 += src->m00 * tx + src->m10 * ty + src->m20 * tz;
116     dst->m31 += src->m01 * tx + src->m11 * ty + src->m21 * tz;
117     dst->m32 += src->m02 * tx + src->m12 * ty + src->m22 * tz;
118     dst->m33 += src->m03 * tx + src->m13 * ty + src->m23 * tz;
119     return (dst);
120 }
121
122 t_mat4f * mat4f_translate3(t_mat4f * dst, t_mat4f * src, t_vec3f * translate) {
123     return (mat4f_translate(dst, src, translate->x, translate->y, translate->z));
124 }
125
```

```
126 t_mat4f * mat4f_rotate(t_mat4f * dst, t_mat4f * src, float angle, t_vec3f * axis) {
127
128     if (dst == NULL) {
129         if ((dst = mat4f_new()) == NULL) {
130             return (NULL);
131         }
132     }
133
134     float c = (float)cos(angle);
135     float s = (float)sin(angle);
136     float oneminusc = 1.0f - c;
137     float xy = axis->x * axis->y;
138     float yz = axis->y * axis->z;
139     float xz = axis->x * axis->z;
140     float xs = axis->x * s;
141     float ys = axis->y * s;
142     float zs = axis->z * s;
143
144     float f00 = axis->x * axis->x * oneminusc + c;
145     float f01 = xy * oneminusc + zs;
146     float f02 = xz * oneminusc - ys;
147     // n[3] not used
148     float f10 = xy * oneminusc - zs;
149     float f11 = axis->y * axis->y * oneminusc + c;
150     float f12 = yz * oneminusc + xs;
151     // n[7] not used
152     float f20 = xz * oneminusc + ys;
153     float f21 = yz * oneminusc - xs;
154     float f22 = axis->z * axis->z * oneminusc + c;
155
156     float t00 = src->m00 * f00 + src->m10 * f01 + src->m20 * f02;
157     float t01 = src->m01 * f00 + src->m11 * f01 + src->m21 * f02;
158     float t02 = src->m02 * f00 + src->m12 * f01 + src->m22 * f02;
159     float t03 = src->m03 * f00 + src->m13 * f01 + src->m23 * f02;
160     float t10 = src->m00 * f10 + src->m10 * f11 + src->m20 * f12;
161     float t11 = src->m01 * f10 + src->m11 * f11 + src->m21 * f12;
162     float t12 = src->m02 * f10 + src->m12 * f11 + src->m22 * f12;
163     float t13 = src->m03 * f10 + src->m13 * f11 + src->m23 * f12;
164     dst->m20 = src->m00 * f20 + src->m10 * f21 + src->m20 * f22;
165     dst->m21 = src->m01 * f20 + src->m11 * f21 + src->m21 * f22;
166     dst->m22 = src->m02 * f20 + src->m12 * f21 + src->m22 * f22;
167     dst->m23 = src->m03 * f20 + src->m13 * f21 + src->m23 * f22;
168     dst->m00 = t00;
169     dst->m01 = t01;
170     dst->m02 = t02;
171     dst->m03 = t03;
172     dst->m10 = t10;
173     dst->m11 = t11;
174     dst->m12 = t12;
175     dst->m13 = t13;
176     return (dst);
177 }
178
179 t_mat4f * mat4f_rotateX(t_mat4f * dst, t_mat4f * src, float angle) {
180
181     if (dst == NULL) {
182         if ((dst = mat4f_new()) == NULL) {
183             return (NULL);
184         }
185     }
186
187     float c = (float)cos(angle);
188     float s = (float)sin(angle);
189     float t00 = src->m00;
190     float t01 = src->m01;
191     float t02 = src->m02;
192     float t03 = src->m03;
193     float t10 = src->m10 * c + src->m20 * s;
194     float t11 = src->m11 * c + src->m21 * s;
195     float t12 = src->m12 * c + src->m22 * s;
196     float t13 = src->m13 * c + src->m23 * s;
197     dst->m20 = src->m10 * -s + src->m20 * c;
198     dst->m21 = src->m11 * -s + src->m21 * c;
199     dst->m22 = src->m12 * -s + src->m22 * c;
200     dst->m23 = src->m13 * -s + src->m23 * c;
201     dst->m00 = t00;
202     dst->m01 = t01;
203     dst->m02 = t02;
204     dst->m03 = t03;
205     dst->m10 = t10;
206     dst->m11 = t11;
207     dst->m12 = t12;
208     dst->m13 = t13;
209
210     return (dst);
211 }
212
```

```
213 t_mat4f * mat4f_rotateY(t_mat4f * dst, t_mat4f * src, float angle) {
214
215     if (dst == NULL) {
216         if ((dst = mat4f_new()) == NULL) {
217             return (NULL);
218         }
219     }
220
221     float c = (float)cos(angle);
222     float s = (float)sin(angle);
223     float t00 = src->m00 * c + src->m20 * -s;
224     float t01 = src->m01 * c + src->m21 * -s;
225     float t02 = src->m02 * c + src->m22 * -s;
226     float t03 = src->m03 * c + src->m23 * -s;
227     float t10 = src->m10;
228     float t11 = src->m11;
229     float t12 = src->m12;
230     float t13 = src->m13;
231     dst->m20 = src->m00 * s + src->m20 * c;
232     dst->m21 = src->m01 * s + src->m21 * c;
233     dst->m22 = src->m02 * s + src->m22 * c;
234     dst->m23 = src->m03 * s + src->m23 * c;
235     dst->m00 = t00;
236     dst->m01 = t01;
237     dst->m02 = t02;
238     dst->m03 = t03;
239     dst->m10 = t10;
240     dst->m11 = t11;
241     dst->m12 = t12;
242     dst->m13 = t13;
243     return (dst);
244 }
245
246 t_mat4f * mat4f_rotateZ(t_mat4f * dst, t_mat4f * src, float angle) {
247     if (dst == NULL) {
248         if ((dst = mat4f_new()) == NULL) {
249             return (NULL);
250         }
251     }
252
253     float c = (float)cos(angle);
254     float s = (float)sin(angle);
255     float t00 = src->m00 * c + src->m10 * s;
256     float t01 = src->m01 * c + src->m11 * s;
257     float t02 = src->m02 * c + src->m12 * s;
258     float t03 = src->m03 * c + src->m13 * s;
259     float t10 = src->m00 * -s + src->m10 * c;
260     float t11 = src->m01 * -s + src->m11 * c;
261     float t12 = src->m02 * -s + src->m12 * c;
262     float t13 = src->m03 * -s + src->m13 * c;
263     dst->m20 = src->m20;
264     dst->m21 = src->m21;
265     dst->m22 = src->m22;
266     dst->m23 = src->m23;
267     dst->m00 = t00;
268     dst->m01 = t01;
269     dst->m02 = t02;
270     dst->m03 = t03;
271     dst->m10 = t10;
272     dst->m11 = t11;
273     dst->m12 = t12;
274     dst->m13 = t13;
275     return (dst);
276 }
277
278 t_mat4f * mat4f_rotateXYZ(t_mat4f * dst, t_mat4f * src, t_vec3f * rot) {
279     if (dst == NULL) {
280         if ((dst = mat4f_new()) == NULL) {
281             return (NULL);
282         }
283     }
284
285     mat4f_rotateX(dst, src, rot->x);
286     mat4f_rotateY(dst, src, rot->y);
287     mat4f_rotateZ(dst, src, rot->z);
288     return (dst);
289 }
290
291 t_mat4f * mat4f_transformation(t_mat4f * dst, t_vec3f * translate, t_vec3f * rot, t_vec3f * scale
    ) {
292
293     if ((dst = mat4f_identity(dst)) == NULL) {
294         return (NULL);
295     }
296
297     mat4f_translate3(dst, dst, translate);
298     mat4f_rotateXYZ(dst, dst, rot);
```

```
299     mat4f_scale3(dst, dst, scale);
300     return (dst);
301 }
302 
303 float mat4f_determinant(t_mat4f * mat) {
304 
305     if (mat == NULL) {
306         return (0);
307     }
308 
309     float d = 0;
310     d += mat->m00 * ((mat->m11 * mat->m22 * mat->m33 + mat->m12 * mat->m23 * mat->m31 + mat->m13
            * mat->m21 * mat->m32)
311         - mat->m13 * mat->m22 * mat->m31 - mat->m11 * mat->m23 * mat->m32 - mat->m12 * mat->m21 *
                mat->m33);
312     d -= mat->m01 * ((mat->m10 * mat->m22 * mat->m33 + mat->m12 * mat->m23 * mat->m30 + mat->m13
            * mat->m20 * mat->m32)
313         - mat->m13 * mat->m22 * mat->m30 - mat->m10 * mat->m23 * mat->m32 - mat->m12 * mat->m20 *
                mat->m33);
314     d += mat->m02 * ((mat->m10 * mat->m21 * mat->m33 + mat->m11 * mat->m23 * mat->m30 + mat->m13
            * mat->m20 * mat->m31)
315         - mat->m13 * mat->m21 * mat->m30 - mat->m10 * mat->m23 * mat->m31 - mat->m11 * mat->m20 *
                mat->m33);
316     d -= mat->m03 * ((mat->m10 * mat->m21 * mat->m32 + mat->m11 * mat->m22 * mat->m30 + mat->m12
            * mat->m20 * mat->m31)
317         - mat->m12 * mat->m21 * mat->m30 - mat->m10 * mat->m22 * mat->m31 - mat->m11 * mat->m20 *
                mat->m32);
318     return (d);
319 }
320 
321 static float _mat4f_determinant3x3(float t00, float t01, float t02, float t10, float t11, float
        t12, float t20, float t21, float t22) {
322     return (t00 * (t11 * t22 - t12 * t21) + t01 * (t12 * t20 - t10 * t22) + t02 * (t10 * t21 -
            t11 * t20));
323 }
324 
325 t_mat4f * mat4f_invert(t_mat4f * dst, t_mat4f * src) {
326 
327     float determinant = mat4f_determinant(src);
328 
329     if (determinant != 0) {
330         if (dst == NULL) {
331             if ((dst = mat4f_new()) == NULL) {
332                 return (NULL);
333             }
334         }
335 
336         float determinant_inv = 1.0f / determinant;
337 
338         float t00 = _mat4f_determinant3x3(src->m11, src->m12, src->m13, src->m21, src->m22, src->
                m23, src->m31, src->m32, src->m33);
339         float t01 = -_mat4f_determinant3x3(src->m10, src->m12, src->m13, src->m20, src->m22, src
                ->m23, src->m30, src->m32, src->m33);
340         float t02 = _mat4f_determinant3x3(src->m10, src->m11, src->m13, src->m20, src->m21, src->
                m23, src->m30, src->m31, src->m33);
341         float t03 = -_mat4f_determinant3x3(src->m10, src->m11, src->m12, src->m20, src->m21, src
                ->m22, src->m30, src->m31, src->m32);
342 
343         float t10 = -_mat4f_determinant3x3(src->m01, src->m02, src->m03, src->m21, src->m22, src
                ->m23, src->m31, src->m32, src->m33);
344         float t11 = _mat4f_determinant3x3(src->m00, src->m02, src->m03, src->m20, src->m22, src->
                m23, src->m30, src->m32, src->m33);
345         float t12 = -_mat4f_determinant3x3(src->m00, src->m01, src->m03, src->m20, src->m21, src
                ->m23, src->m30, src->m31, src->m33);
346         float t13 = _mat4f_determinant3x3(src->m00, src->m01, src->m02, src->m20, src->m21, src->
                m22, src->m30, src->m31, src->m32);
347 
348         float t20 = _mat4f_determinant3x3(src->m01, src->m02, src->m03, src->m11, src->m12, src->
                m13, src->m31, src->m32, src->m33);
349         float t21 = -_mat4f_determinant3x3(src->m00, src->m02, src->m03, src->m10, src->m12, src
                ->m13, src->m30, src->m32, src->m33);
350         float t22 = _mat4f_determinant3x3(src->m00, src->m01, src->m03, src->m10, src->m11, src->
                m13, src->m30, src->m31, src->m33);
351         float t23 = -_mat4f_determinant3x3(src->m00, src->m01, src->m02, src->m10, src->m11, src
                ->m12, src->m30, src->m31, src->m32);
352 
353         float t30 = -_mat4f_determinant3x3(src->m01, src->m02, src->m03, src->m11, src->m12, src
                ->m13, src->m21, src->m22, src->m23);
354         float t31 = _mat4f_determinant3x3(src->m00, src->m02, src->m03, src->m10, src->m12, src->
                m13, src->m20, src->m22, src->m23);
355         float t32 = -_mat4f_determinant3x3(src->m00, src->m01, src->m03, src->m10, src->m11, src
                ->m13, src->m20, src->m21, src->m23);
356         float t33 = _mat4f_determinant3x3(src->m00, src->m01, src->m02, src->m10, src->m11, src->
                m12, src->m20, src->m21, src->m22);
357 
358         // transpose and divide by the determinant
359         dst->m00 = t00 * determinant_inv;
```

```
360         dst->m11 = t11 * determinant_inv;
361         dst->m22 = t22 * determinant_inv;
362         dst->m33 = t33 * determinant_inv;
363         dst->m01 = t10 * determinant_inv;
364         dst->m10 = t01 * determinant_inv;
365         dst->m20 = t02 * determinant_inv;
366         dst->m02 = t20 * determinant_inv;
367         dst->m12 = t21 * determinant_inv;
368         dst->m21 = t12 * determinant_inv;
369         dst->m03 = t30 * determinant_inv;
370         dst->m30 = t03 * determinant_inv;
371         dst->m13 = t31 * determinant_inv;
372         dst->m31 = t13 * determinant_inv;
373         dst->m32 = t23 * determinant_inv;
374         dst->m23 = t32 * determinant_inv;
375         return (dst);
376     }
377 
378     return (NULL);
379 }
380 
381 t_mat4f * mat4f_mult(t_mat4f * dst, t_mat4f * left, t_mat4f * right) {
382 
383     if (dst == NULL) {
384         if ((dst = mat4f_new()) == NULL) {
385             return (NULL);
386         }
387     }
388 
389     float m00 = left->m00 * right->m00 + left->m10 * right->m01 + left->m20 * right->m02 + left->
        m30 * right->m03;
390     float m01 = left->m01 * right->m00 + left->m11 * right->m01 + left->m21 * right->m02 + left->
        m31 * right->m03;
391     float m02 = left->m02 * right->m00 + left->m12 * right->m01 + left->m22 * right->m02 + left->
        m32 * right->m03;
392     float m03 = left->m03 * right->m00 + left->m13 * right->m01 + left->m23 * right->m02 + left->
        m33 * right->m03;
393     float m10 = left->m00 * right->m10 + left->m10 * right->m11 + left->m20 * right->m12 + left->
        m30 * right->m13;
394     float m11 = left->m01 * right->m10 + left->m11 * right->m11 + left->m21 * right->m12 + left->
        m31 * right->m13;
395     float m12 = left->m02 * right->m10 + left->m12 * right->m11 + left->m22 * right->m12 + left->
        m32 * right->m13;
396     float m13 = left->m03 * right->m10 + left->m13 * right->m11 + left->m23 * right->m12 + left->
        m33 * right->m13;
397     float m20 = left->m00 * right->m20 + left->m10 * right->m21 + left->m20 * right->m22 + left->
        m30 * right->m23;
398     float m21 = left->m01 * right->m20 + left->m11 * right->m21 + left->m21 * right->m22 + left->
        m31 * right->m23;
399     float m22 = left->m02 * right->m20 + left->m12 * right->m21 + left->m22 * right->m22 + left->
        m32 * right->m23;
400     float m23 = left->m03 * right->m20 + left->m13 * right->m21 + left->m23 * right->m22 + left->
        m33 * right->m23;
401     float m30 = left->m00 * right->m30 + left->m10 * right->m31 + left->m20 * right->m32 + left->
        m30 * right->m33;
402     float m31 = left->m01 * right->m30 + left->m11 * right->m31 + left->m21 * right->m32 + left->
        m31 * right->m33;
403     float m32 = left->m02 * right->m30 + left->m12 * right->m31 + left->m22 * right->m32 + left->
        m32 * right->m33;
404     float m33 = left->m03 * right->m30 + left->m13 * right->m31 + left->m23 * right->m32 + left->
        m33 * right->m33;
405 
406     dst->m00 = m00;
407     dst->m01 = m01;
408     dst->m02 = m02;
409     dst->m03 = m03;
410     dst->m10 = m10;
411     dst->m11 = m11;
412     dst->m12 = m12;
413     dst->m13 = m13;
414     dst->m20 = m20;
415     dst->m21 = m21;
416     dst->m22 = m22;
417     dst->m23 = m23;
418     dst->m30 = m30;
419     dst->m31 = m31;
420     dst->m32 = m32;
421     dst->m33 = m33;
422 
423     return (dst);
424 }
425 
426 t_vec4f * mat4f_transform_vec4f(t_vec4f * dst, t_mat4f * left, t_vec4f * right) {
427     if (dst == NULL) {
428         if ((dst = vec4f_new()) == NULL) {
429             return (NULL);
430         }
```

```
431     }
432
433     float x = left->m00 * right->x + left->m10 * right->y + left->m20 * right->z + left->m30 *
            right->w;
434     float y = left->m01 * right->x + left->m11 * right->y + left->m21 * right->z + left->m31 *
            right->w;
435     float z = left->m02 * right->x + left->m12 * right->y + left->m22 * right->z + left->m32 *
            right->w;
436     float w = left->m03 * right->x + left->m13 * right->y + left->m23 * right->z + left->m33 *
            right->w;
437
438     dst->x = x;
439     dst->y = y;
440     dst->z = z;
441     dst->w = w;
442
443     return (dst);
444 }
445
446 t_mat4f * mat4f_orthographic(t_mat4f * dst, float left, float right, float bot, float top, float
        near, float far) {
447     if (dst == NULL) {
448         if ((dst = mat4f_new()) == NULL) {
449             return (NULL);
450         }
451     }
452
453     dst->m00 = 2.0f / (right - left);
454     dst->m01 = 0;
455     dst->m02 = 0;
456     dst->m03 = (right + left) / (left - right);
457
458     dst->m10 = 0;
459     dst->m11 = 2.0f / (top - bot);
460     dst->m12 = 0;
461     dst->m13 = (top + bot) / (bot - top);
462
463     dst->m20 = 0;
464     dst->m21 = 0;
465     dst->m22 = 2 / (near - far);
466     dst->m23 = (far + near) / (near - far);
467
468     dst->m30 = 0.0f;
469     dst->m31 = 0.0f;
470     dst->m32 = 0.0f;
471     dst->m33 = 1.0f;
472
473     return (dst);
474 }
475
476 t_mat4f * mat4f_perspective(t_mat4f * dst, float aspect, float fov, float near, float far) {
477     if (dst == NULL) {
478         if ((dst = mat4f_new()) == NULL) {
479             return (NULL);
480         }
481     }
482
483     float y_scale = (float) (1.0f / tan(fov / 2.0f) * aspect);
484     float x_scale = y_scale / aspect;
485     float frustrum_length = far - near;
486
487     dst->m00 = x_scale;
488     dst->m01 = 0.0f;
489     dst->m02 = 0.0f;
490     dst->m03 = 0.0f;
491
492     dst->m10 = 0.0f;
493     dst->m11 = y_scale;
494     dst->m12 = 0.0f;
495     dst->m13 = 0.0f;
496
497     dst->m20 = 0.0f;
498     dst->m21 = 0.0f;
499     dst->m22 = -((far + near) / frustrum_length);
500     dst->m23 = -1.0f;
501
502     dst->m30 = 0.0f;
503     dst->m31 = 0.0f;
504     dst->m32 = -((2.0f * near * far) / frustrum_length);
505     dst->m33 = 0.0f;
506
507     return (dst);
508 }
509
510 char * mat4f_str(t_mat4f * mat) {
511     if (mat == NULL) {
512         return (strdup("mat4f(NULL)"));
```

```
513     }
514     char buffer[1024];
515     sprintf(buffer, "mat4f(%.4f ; %.4f ; %.4f ; %.4f\n      %.4f ; %.4f ; %.4f ; %.4f\n      %.4f
            ; %.4f ; %.4f ; %.4f\n      %.4f ; %.4f ; %.4f ; %.4f)",
516         mat->m00, mat->m10, mat->m20, mat->m30, mat->m01, mat->m11, mat->m21, mat->m31, mat->m02,
                mat->m12, mat->m22, mat->m32, mat->m03, mat->m13, mat->m23, mat->m33);
517     return (strdup(buffer));
518 }
```

maths/srcs/vec2i.c

```
1   #include "vec2i.h"
2
3   t_vec2i * vec2i_new(void) {
4       return ((t_vec2i *)malloc(sizeof(t_vec2i)));
5   }
6
7   void vec2i_delete(t_vec2i * vec) {
8       free(vec);
9   }
10
11  t_vec2i * vec2i_zero(t_vec2i * dst) {
12      if (dst == NULL) {
13          if ((dst = vec2i_new()) == NULL) {
14              return (NULL);
15          }
16      }
17      memset(dst, 0, sizeof(t_vec2i));
18      return (dst);
19  }
20
21  t_vec2i * vec2i_set(t_vec2i * dst, int x, int y) {
22      if (dst == NULL) {
23          if ((dst = vec2i_new()) == NULL) {
24              return (NULL);
25          }
26      }
27      dst->x = x;
28      dst->y = y;
29      return (dst);
30  }
31
32  t_vec2i * vec2i_set2(t_vec2i * dst, t_vec2i * vec) {
33      if (dst == vec) {
34          return (dst);
35      }
36      return (vec2i_set(dst, vec->x, vec->y));
37  }
38
39  t_vec2i * vec2i_add(t_vec2i * dst, t_vec2i * left, t_vec2i * right) {
40      if (dst == NULL) {
41          if ((dst = vec2i_new()) == NULL) {
42              return (NULL);
43          }
44      }
45      dst->x = left->x + right->x;
46      dst->y = left->y + right->y;
47      return (dst);
48  }
49
50  t_vec2i * vec2i_sub(t_vec2i * dst, t_vec2i * left, t_vec2i * right) {
51      if (dst == NULL) {
52          if ((dst = vec2i_new()) == NULL) {
53              return (NULL);
54          }
55      }
56      dst->x = left->x - right->x;
57      dst->y = left->y - right->y;
58      return (dst);
59  }
60
61  t_vec2i * vec2i_mult(t_vec2i * dst, t_vec2i * vec, int scalar) {
62      if (dst == NULL) {
63          if ((dst = vec2i_new()) == NULL) {
64              return (NULL);
65          }
66      }
67      dst->x = vec->x * scalar;
68      dst->y = vec->y * scalar;
69      return (dst);
70  }
71
72  t_vec2i * vec2i_mult2(t_vec2i * dst, t_vec2i * left, t_vec2i * right) {
73      if (dst == NULL) {
```

```
74          if ((dst = vec2i_new()) == NULL) {
75              return (NULL);
76          }
77      }
78      dst->x = left->x * right->x;
79      dst->y = left->y * right->y;
80      return (dst);
81  }
82
83  int vec2i_dot_product(t_vec2i * left, t_vec2i * right) {
84      return (left->x * right->x + left->y * right->y);
85  }
86
87  int vec2i_length_squared(t_vec2i * vec) {
88      return (vec2i_dot_product(vec, vec));
89  }
90
91  int vec2i_length(t_vec2i * vec) {
92      return ((int)sqrt(vec2i_length_squared(vec)));
93  }
94
95  t_vec2i * vec2i_normalize(t_vec2i * dst, t_vec2i * vec) {
96
97      if (dst == NULL) {
98          if ((dst = vec2i_new()) == NULL) {
99              return (NULL);
100         }
101     }
102
103     int norm = 1 / vec2i_length(vec);
104     dst->x = vec->x * norm;
105     dst->y = vec->y * norm;
106     return (dst);
107 }
108
109 t_vec2i * vec2i_negate(t_vec2i * dst, t_vec2i * src) {
110     if (dst == NULL) {
111         if ((dst = vec2i_new()) == NULL) {
112             return (NULL);
113         }
114     }
115     dst->x = -src->x;
116     dst->y = -src->y;
117     return (dst);
118 }
119
120 int vec2i_angle(t_vec2i * left, t_vec2i * right) {
121     int dls = vec2i_dot_product(left, right) / (vec2i_length(left) * vec2i_length(right));
122     if (dls < -1.0f) {
123         dls = -1.0f;
124     } else if (dls > 1.0f) {
125         dls = 1.0f;
126     }
127     return ((int)acos(dls));
128 }
129
130 t_vec2i * vec2i_mix(t_vec2i * dst, t_vec2i * left, t_vec2i * right, int ratio) {
131
132     if (dst == NULL) {
133         if ((dst = vec2i_new()) == NULL) {
134             return (NULL);
135         }
136     }
137
138     dst->x = left->x * ratio + right->x * (1 - ratio);
139     dst->y = left->y * ratio + right->y * (1 - ratio);
140     return (dst);
141 }
142
143 int vec2i_equals(t_vec2i * left, t_vec2i * right) {
144     return (left == right || (left->x == right->x && left->y == right->y));
145 }
146
147 char * vec2i_str(t_vec2i * vec) {
148     if (vec == NULL) {
149         return (strdup("vec2i(NULL)"));
150     }
151     char buffer[128];
152     sprintf(buffer, "vec2i(%d ; %d)", vec->x, vec->y);
153     return (strdup(buffer));
154 }
```

maths/srcs/vec3f.c

```c
#include "vec3f.h"

t_vec3f * vec3f_new(void) {
    return ((t_vec3f *)malloc(sizeof(t_vec3f)));
}

void vec3f_delete(t_vec3f * vec) {
    free(vec);
}

t_vec3f * vec3f_zero(t_vec3f * dst) {
    if (dst == NULL) {
        if ((dst = vec3f_new()) == NULL) {
            return (NULL);
        }
    }
    memset(dst, 0, sizeof(t_vec3f));
    return (dst);
}

t_vec3f * vec3f_set(t_vec3f * dst, float x, float y, float z) {
    if (dst == NULL) {
        if ((dst = vec3f_new()) == NULL) {
            return (NULL);
        }
    }
    dst->x = x;
    dst->y = y;
    dst->z = z;
    return (dst);
}

t_vec3f * vec3f_set3(t_vec3f * dst, t_vec3f * vec) {
    if (dst == vec) {
        return (dst);
    }
    return (vec3f_set(dst, vec->x, vec->y, vec->z));
}

t_vec3f * vec3f_add(t_vec3f * dst, t_vec3f * left, t_vec3f * right) {
    if (dst == NULL) {
        if ((dst = vec3f_new()) == NULL) {
            return (NULL);
        }
    }
    dst->x = left->x + right->x;
    dst->y = left->y + right->y;
    dst->z = left->z + right->z;
    return (dst);
}

t_vec3f * vec3f_sub(t_vec3f * dst, t_vec3f * left, t_vec3f * right) {
    if (dst == NULL) {
        if ((dst = vec3f_new()) == NULL) {
            return (NULL);
        }
    }
    dst->x = left->x - right->x;
    dst->y = left->y - right->y;
    dst->z = left->z - right->z;
    return (dst);
}

t_vec3f * vec3f_mult(t_vec3f * dst, t_vec3f * vec, float scalar) {
    if (dst == NULL) {
        if ((dst = vec3f_new()) == NULL) {
            return (NULL);
        }
    }
    dst->x = vec->x * scalar;
    dst->y = vec->y * scalar;
    dst->z = vec->z * scalar;
    return (dst);
}

t_vec3f * vec3f_mult3(t_vec3f * dst, t_vec3f * left, t_vec3f * right) {
    if (dst == NULL) {
        if ((dst = vec3f_new()) == NULL) {
            return (NULL);
        }
    }
    dst->x = left->x * right->x;
    dst->y = left->y * right->y;
    dst->z = left->z * right->z;
    return (dst);
}
```

```
87
88  /** cross product */
89  t_vec3f * vec3f_cross(t_vec3f * dst, t_vec3f * left, t_vec3f * right) {
90      if (dst == NULL) {
91          if ((dst = vec3f_new()) == NULL) {
92              return (NULL);
93          }
94      }
95
96      dst->x = left->y * right->z - left->z * right->y;
97      dst->y = left->z * right->x - left->x * right->z;
98      dst->z = left->x * right->y - left->y * right->x;
99      return (dst);
100 }
101
102 float vec3f_dot_product(t_vec3f * left, t_vec3f * right) {
103     return (left->x * right->x + left->y * right->y + left->z * right->z);
104 }
105
106 float vec3f_length_squared(t_vec3f * vec) {
107     return (vec3f_dot_product(vec, vec));
108 }
109
110 float vec3f_length(t_vec3f * vec) {
111     return ((float)sqrt(vec3f_length_squared(vec)));
112 }
113
114 t_vec3f * vec3f_normalize(t_vec3f * dst, t_vec3f * vec) {
115
116     if (dst == NULL) {
117         if ((dst = vec3f_new()) == NULL) {
118             return (NULL);
119         }
120     }
121
122     float norm = 1 / vec3f_length(vec);
123     dst->x = vec->x * norm;
124     dst->y = vec->y * norm;
125     dst->z = vec->z * norm;
126     return (dst);
127 }
128
129 t_vec3f * vec3f_negate(t_vec3f * dst, t_vec3f * src) {
130     if (dst == NULL) {
131         if ((dst = vec3f_new()) == NULL) {
132             return (NULL);
133         }
134     }
135     dst->x = -src->x;
136     dst->y = -src->y;
137     dst->z = -src->z;
138     return (dst);
139 }
140
141 float vec3f_angle(t_vec3f * left, t_vec3f * right) {
142     float dls = vec3f_dot_product(left, right) / (vec3f_length(left) * vec3f_length(right));
143     if (dls < -1.0f) {
144         dls = -1.0f;
145     } else if (dls > 1.0f) {
146         dls = 1.0f;
147     }
148     return ((float)acos(dls));
149 }
150
151 t_vec3f * vec3f_mix(t_vec3f * dst, t_vec3f * left, t_vec3f * right, float ratio) {
152
153     if (dst == NULL) {
154         if ((dst = vec3f_new()) == NULL) {
155             return (NULL);
156         }
157     }
158
159     dst->x = left->x * ratio + right->x * (1 - ratio);
160     dst->y = left->y * ratio + right->y * (1 - ratio);
161     dst->z = left->z * ratio + right->z * (1 - ratio);
162     return (dst);
163 }
164
165 int vec3f_equals(t_vec3f * left, t_vec3f * right) {
166     return (left == right || (left->x == right->x && left->y == right->y && left->z == right->z))
        ;
167 }
168
169 char * vec3f_str(t_vec3f * vec) {
170     if (vec == NULL) {
171         return (strdup("vec3f(NULL)"));
172     }
```

```
173     char buffer[160];
174     sprintf(buffer, "vec3f(%f ; %f ; %f)", vec->x, vec->y, vec->z);
175     return (strdup(buffer));
176 }
```

3 Listes et table de hashage

data_structures/includes/array_list.h

```
1   #ifndef ARRAY_LIST_H
2   # define ARRAY_LIST_H
3
4   # include "common.h"
5   # include <string.h>
6   # include <stdlib.h>
7   # include <stdio.h>
8
9   typedef struct  s_array_list {
10      char                * data;
11      unsigned long int   capacity;
12      unsigned long int   size;
13      unsigned int        elem_size;
14      unsigned int        default_capacity;
15  }               t_array_list;
16
17  /**
18   * Create a new array list
19   * nb : number of elements which the array can hold on first allocation
20   * elem_size : size of an elements
21   *
22   * e.g: t_array_list array = array_list_new(16, sizeof(int));
23   */
24  t_array_list * array_list_new(unsigned long int nb, unsigned int elem_size);
25
26  /** Add an element at the end of the list */
27  int array_list_add(t_array_list * array, void * data);
28
29  /** Clear the list (remove every data, and resize it to the default capacity) */
30  void array_list_clear(t_array_list * array);
31
32  /**
33   *  Delete DEFINETELY the list from memory
34   */
35  void array_list_delete(t_array_list * array);
36
37  /** remove the element at given index */
38  void array_list_remove(t_array_list * array, unsigned int idx);
39
40  /**
41   *  Sort the array list using std quicksort algorythm
42   *
43   *  e.g:    t_array_list array = array_list_new(16, sizeof(char) * 2);
44   *          array_list_push(&array, "d");
45   *          array_list_push(&array, "a");
46   *          array_list_push(&array, "f");
47   *          [...]
48   *          array_list_sort(&array, (t_cmp_function)strcmp);
49   */
50  void array_list_sort(t_array_list * array, t_cmp_function cmpf);
51
52  /**
53   *  Add every elements the end of the list
54   *  this function is faster than calling multiples 'array_list_add()'
55   *  so consider using it :)
56   */
57  void array_list_add_all(t_array_list * array, void * buffer, unsigned long int nb);
58
59  /**
60   *  Get raw data of your array list
61   *  (buffer of every data)
62   *  You should really not use this function
63   */
64  void * array_list_raw(t_array_list * array);
65
66  /** get item by index */
67  void * array_list_get(t_array_list * array, unsigned int idx);
68
69  /**
70   * Iterate on the array list using a macro
71   *
72   * i.e :    t_array_list array;
73   *
```

```
 *          [...] //push strings to the list
 *
 *          // print every string which the array list holds
 *          ARRAY_LIST_ITER_START(array, char *, str, i)
 *          {
 *              puts(str);
 *          }
 *          ARRAY_LIST_ITER_END(array, char *, str, i);
 */
# define ARRAY_LIST_ITER_START(L, T, X, I)\
 {\
    unsigned long int I = 0;\
    while (I < (L)->size) {\
        T X = ((T)(L)->data) + I;
# define ARRAY_LIST_ITER_END(L, T, X, I)\
       ++I;\
    }\
}

#endif
```

data_structures/includes/common.h

```
#ifndef COMMON_H
# define COMMON_H

# include <sys/time.h>
# include <stdlib.h>
# include <string.h>
# include <stdio.h>

typedef void (*t_function)();
typedef int (*t_cmp_function) (void const * a, void const * b);
typedef unsigned long int (*t_hash_function) (void const * v);


typedef t_function t_f;
typedef t_cmp_function t_cmpf;
typedef t_hash_function t_hf;

# define MICROSEC(V)    {\
    struct timeval tv;\
    gettimeofday(&tv, NULL);\
    V = 1000000 * tv.tv_sec + tv.tv_usec;\
}


#endif
```

data_structures/includes/hmap.h

```
#ifndef HMAP_H
# define HMAP_H

# include "common.h"
# include "linked_list.h"

/**
 *  Generic hash map implementation in C89:
 *
 *  ABOUT THE IMPLEMENTATION:
 *      - given pointer address are saved for values. No copy their data are done. (same for keys)
 *      - const where used where on constant data (well...), so you dont mess up the hash map :)
 *      - an array of linked list is used to handle collisions
 *
 *
 *  example for a string hashmap:
 *
 *      t_hmap map = hmap_new(1024, (t_hf)strhash, (t_cmpf)strcmp);
 *      hmap_insert(&map, strdup("hello world"), strdup("im a key"), strlen("Hello world") + 1);
 *      char *helloworld = hmap_get(&map, "im a key"); //now contains "Hello world"
 */

typedef struct  s_hmap_node {
    unsigned long int const hash; //hash of the key
    void const * data; //the data holds
    void const * key; //the key used
}               t_hmap_node;

```

```c
typedef struct  s_hmap {
    t_list * values; //a buffer of value holders (to handle collision)
    unsigned long int capacity; //number of lists
    unsigned long int size; //number of value set
    t_hash_function hashf; //hash function
    t_cmp_function keycmpf; //key comparison function, where node keys are sent as parameters
    t_function datafreef; //function call when a data object should be freed
    t_function keyfreef; //function called when a key should be freed
}               t_hmap;

/**
 *  Create a new hashmap:
 *
 *  capacity : capacity of the hashmap (number of lists boxes in memory)
 *  hashf    : hash function to use on inserted elements
 *  cmpf     : comparison function to use when searching a data
 */
t_hmap * hmap_new(unsigned long int const capacity, t_hash_function hashf, t_cmp_function keycmpf
    , t_function keyfreef, t_function datafreef);

/**
 *  Delete the hashmap from the heap
 *
 *  hmap : hash map
 *  datafreef : function which will be called on node data before the node being freed.
 *              i.e :   'NULL' if data shouldnt be free, 'free' if the data was allocated with a
     malloc,
 *                      'myfree' if this is structure which contains multiple allocated fields
     ...
 *  keyfreef : same for the node key
 */
void hmap_delete(t_hmap * hmap);

/**
 *  Insert a value into the hashmap:
 *
 *  map  : hmap
 *  data : value to insert
 *  key  : key reference for this data
 *  size : size of the data (i.e, 'sizeof(t_data_structure)', 'strlen(str) + 1')
 *
 *  return the given data if it was inserted properly, NULL elseway
 */
void const * hmap_insert(t_hmap * hmap, void const * data, void const * key);

/**
 *  Get data from the hashmap
 *
 *  hmap : hash map
 *  key  : the node's key to find
 */
void * hmap_get(t_hmap * hmap, void const * key);

/**
 *  Remove the data pointer from the hash map
 *  return 1 if the element was removed, 0 elseway
 *  hmap : the hash map
 *  data : pointer to the data
 */
int hmap_remove_data(t_hmap * hmap, void const * data);

/**
 *  Remove the data which match with the given key from the hash map
 *  return 1 if the element was removed, 0 elseway
 *
 *  hmap : the hash map
 *  key  : pointer to the key
 */
int hmap_remove_key(t_hmap * hmap, void const * key);

/**
 *  Some simple builtin hashes functions, useful for tests.
 *
 *  String hash is based on : http://www.cse.yorku.ca/~oz/hash.html
 */
unsigned long int strhash(char const * str);
unsigned long int inthash(int const value);

/**
 *  Macro to iterate fastly though to hash map
 *
 *  i.e:
 *      HMAP_ITER_START(hmap, char *, str) {
 *          puts(str);
 *      }
 *      HMAP_ITER_END(hmap, char *, str)
 */
```

```
# define HMAP_ITER_START(H, T, V)\
{\
    unsigned long int i = 0;\
    while (i < (H)->capacity) {\
        t_list * lst = (H)->values + i;\
        if (lst != NULL && lst->head != NULL) {\
            LIST_ITER_START(lst, t_hmap_node *, node) {\
                T V = (T)(node->data);
# define HMAP_ITER_END(H, T, V)\
            }\
            LIST_ITER_END(lst, t_hmap_node *, node)\
        }\
        ++i;\
    }\
}

#endif
```

data_structures/includes/linked_list.h

```
#ifndef LINKED_LIST_H
# define LINKED_LIST_H

# include <stdlib.h>
# include <string.h>
# include <unistd.h>
# include "common.h"

typedef struct  s_list_node {
    struct s_list_node * next;
    struct s_list_node * prev;
}               t_list_node;

typedef struct  s_list {
    t_list_node         * head;
    unsigned long int   size;
}               t_list;

/** initialize the given list */
int list_init(t_list * list);

/** Create a new linked list */
t_list * list_new(void);

/** Add an element at the end of the list */
void * list_add(t_list * lst, void const * content, unsigned int content_size);

/** Add an element in head of the list */
void * list_addfront(t_list * lst, void const * content, unsigned int content_size);

/**
 *  Return the list node data which match with the given comparison function
 *  and reference data. (cmpf should acts like 'strcmp()')
 */
void * list_get(t_list * lst, t_cmp_function cmpf, void * cmpd);

/**
 * Remove the node which datas match with the given comparison function
 * and the given data reference
 */
int list_remove(t_list * lst, t_cmp_function cmpf, void * cmpref);

/** remove the given node from the list */
void list_remove_node(t_list * lst, t_list_node *node);

/** Remove first / last element of the list. Return 1 if it was removed, 0 else */
int list_remove_first(t_list * lst);
int list_remove_last(t_list * lst);

/** Remove the first element of the list, and return it data */
void * list_pop(t_list * lst);

/** Return the first element of the list */
void * list_head(t_list * lst);

/** Clear the list (remove every node) */
void list_clear(t_list * lst);

/** remove the list for the heap */
void list_delete(t_list * lst);

/** iterate the function to every node content of the list */
void list_iterate(t_list * lst, t_function f);

```

```
65
66  /** Return a buffer which holds pointers to every elements of the list, allocated with 'malloc()'
        */
67  void * list_buffer(t_list * lst);
68
69
70  /** iterate on the list using a macro (optimized) */
71  # define LIST_ITER_START(L, T, V)\
72  {\
73      if (L != NULL && L->head != NULL) {\
74          t_list_node *__node = L->head->next;\
75          while (__node != L->head) {\
76              T V = (T)(__node + 1);
77  # define LIST_ITER_END(L, T, V) \
78              __node = __node->next; \
79          }\
80      }\
81  }
82
83  #endif
```

data_structures/srcs/array_list.c

```
1   #include "array_list.h"
2
3   t_array_list * array_list_new(unsigned long int nb, unsigned int elem_size) {
4       t_array_list * array = (t_array_list *)malloc(sizeof(t_array_list));
5       if (array == NULL) {
6           return (NULL);
7       }
8
9       array->data = calloc(nb, elem_size);
10      array->capacity = nb;
11      array->elem_size = elem_size;
12      array->size = 0;
13      array->default_capacity = nb;
14      return (array);
15  }
16
17  static void array_list_resize(t_array_list * array, unsigned size) {
18      array->data = realloc(array->data, size * array->elem_size);
19      array->capacity = size;
20      if (array->size > size) {
21          array->size = size;
22      }
23  }
24
25  static void array_list_expand(t_array_list * array) {
26      unsigned long int size = array->capacity * 2;
27      array_list_resize(array, size);
28  }
29
30  int array_list_add(t_array_list * array, void * data) {
31      if (array->size == array->capacity) {
32          array_list_expand(array);
33      }
34      memcpy(array->data + array->size * array->elem_size, data, array->elem_size);
35      array->size++;
36      return (array->size);
37  }
38
39  void array_list_add_all(t_array_list * array, void * buffer, unsigned long int nb) {
40      unsigned int array_idx = array->size * array->elem_size;
41      while (nb) {
42          unsigned int copy_nb = array->capacity - array->size;
43          if (copy_nb > nb) {
44              copy_nb = nb;
45          }
46          if (copy_nb == 0) {
47              array_list_expand(array);
48              continue ;
49          }
50          unsigned int copy_size = copy_nb * array->elem_size;
51          memcpy(array->data + array_idx, buffer, copy_size);
52          nb -= copy_nb;
53          array->size += copy_nb;
54          buffer += copy_size;
55          array_idx += copy_size;
56      }
57  }
58
59  void * array_list_get(t_array_list * array, unsigned int idx) {
60      return (array->data + idx * array->elem_size);
61  }
```

```
62
63  void array_list_remove(t_array_list * array, unsigned int idx) {
64      if (array->size == 0 || idx >= array->size) {
65          return ;
66      }
67
68      unsigned int begin = idx * array->elem_size;
69      unsigned int end = (array->size - 1) * array->elem_size;
70      memmove(array->data + begin, array->data + begin + array->elem_size, end - begin);
71
72      array->size--;
73  }
74
75  void array_list_clear(t_array_list * array) {
76      array->size = 0;
77      array_list_resize(array, array->default_capacity);
78  }
79
80  void array_list_delete(t_array_list * array) {
81      free(array->data);
82  }
83
84  void array_list_sort(t_array_list * array, t_cmp_function cmpf) {
85      qsort(array->data, array->size, array->elem_size, cmpf);
86  }
87
88  void * array_list_raw(t_array_list * array) {
89      return (array->data);
90  }
```

data_structures/srcs/hmap.c

```
1   #include "hmap.h"
2
3   t_hmap * hmap_new(unsigned long int const capacity,
4           t_hash_function hashf, t_cmp_function keycmpf,
5           t_function keyfreef, t_function datafreef) {
6
7       // set the hmap capacity to the closest power of two
8       unsigned long int c = 1;
9       while (c < capacity) {
10          c = c << 1;
11      }
12
13      unsigned long int size = sizeof(t_list) * c;
14      void * values = malloc(size);
15      if (values == NULL) {
16          return (NULL);
17      }
18      memset(values, 0, size);
19
20      t_hmap * hmap = (t_hmap *)malloc(sizeof(t_hmap));
21      if (hmap == NULL) {
22          free(values);
23          return (NULL);
24      }
25
26      hmap->values = values;
27      hmap->capacity = capacity;
28      hmap->size = 0;
29      hmap->hashf = hashf;
30      hmap->keycmpf = keycmpf;
31      hmap->datafreef = datafreef;
32      hmap->keyfreef = keyfreef;
33
34      return (hmap);
35  }
36
37  void hmap_delete(t_hmap * hmap) {
38      unsigned long int i = 0;
39      while (i < hmap->capacity) {
40          t_list * lst = hmap->values + i;
41          //if the list has been initialized
42          if (lst->head) {
43              LIST_ITER_START(lst, t_hmap_node *, node) {
44                  if (hmap->datafreef) {
45                      hmap->datafreef(node->data);
46                  }
47
48                  if (hmap->keyfreef) {
49                      hmap->keyfreef(node->key);
50                  }
51              }
52              LIST_ITER_END(lst, t_hmap_node *, node)
```

```
53              list_delete(lst);
54          }
55          ++i;
56      }
57  }
58
59  void const * hmap_insert(t_hmap * hmap, void const * data, void const * key)
60  {
61      unsigned long int hash = hmap->hashf(key); //get the hash for this key
62      unsigned long int addr = hash & (hmap->capacity - 1); //get the array list from the hash
63
64      t_hmap_node node = {hash, data, key}; //set the node buffer
65
66      t_list * lst = hmap->values + addr; //get the list from it address
67      //if the list hasnt already been initialized
68      if (lst->head == NULL) {
69          list_init(lst); //initialize it
70      }
71      list_add(lst, &node, sizeof(t_hmap_node)); //add the node to the list
72
73      hmap->size++;
74      return (data); //return the data
75  }
76
77  void * hmap_get(t_hmap * hmap, void const * key) {
78      unsigned long int hash = hmap->hashf(key); //get the hash for this key
79      unsigned long int addr = hash & (hmap->capacity - 1); //get the lst list from the hash
80
81      t_list * lst = hmap->values + addr; //list of collision for this key hash
82
83      if (lst->size == 0) {
84          return (NULL);
85      }
86
87      //so compare the exact key to find the wanted data
88      LIST_ITER_START(lst, t_hmap_node *, node) {
89          if (hmap->keycmpf(key, node->key) == 0) {
90              return ((void *)node->data);
91          }
92      }
93      LIST_ITER_END(lst, t_hmap_node *, node)
94      return (NULL);
95  }
96
97  int hmap_remove_data(t_hmap * hmap, void const * data) {
98      unsigned long int i = 0;
99      while (i < hmap->capacity) {
100         t_list * lst = hmap->values + i;
101         LIST_ITER_START(lst, t_hmap_node *, node) {
102             if (node->data == data) {
103                 //__node is the current LIST_ITER_START node of the linked list
104                 list_remove_node(lst, __node);
105                 hmap->size--;
106
107                 if (hmap->datafreef) {
108                     hmap->datafreef(node->key);
109                 }
110
111                 if (hmap->keyfreef) {
112                     hmap->keyfreef(node->key);
113                 }
114
115                 return (1);
116             }
117         }
118         LIST_ITER_END(array, t_hmap_node *, node)
119         ++i;
120     }
121     return (0);
122 }
123
124 int hmap_remove_key(t_hmap * hmap, void const * key) {
125     unsigned long int hash = hmap->hashf(key); //get the hash for this key
126     unsigned long int addr = hash & (hmap->capacity - 1); //get the array list from the hash
127
128     t_list * lst = hmap->values + addr; //lst of collision for this key hash
129
130     if (lst->size == 0) {
131         return (0);
132     }
133
134     //so compare the exact key to find the wanted data
135     LIST_ITER_START(lst, t_hmap_node *, node) {
136         if (hmap->keycmpf(key, node->key) == 0) {
137             //__node is the current LIST_ITER_START node of the linked list
138             list_remove_node(lst, __node);
139             hmap->size--;
```

```
140
141             if (hmap->datafreef) {
142                 hmap->datafreef(node->key);
143             }
144
145             if (hmap->keyfreef) {
146                 hmap->keyfreef(node->key);
147             }
148
149             return (1);
150         }
151     }
152     LIST_ITER_END(array, t_hmap_node *, node)
153     return (0);
154 }
155
156 unsigned long int strhash(char const * str) {
157     if (str == NULL) {
158         return (0);
159     }
160
161     unsigned long int hash = 5381;
162     int c;
163     while ((c = *str) != '\0') {
164         hash = ((hash << 5) + hash) + c;
165         str++;
166     }
167     return (hash);
168 }
169
170 unsigned long int inthash(int const value) {
171     return (value);
172 }
```

data_structures/srcs/linked_list.c

```
1  #include "linked_list.h"
2
3  int list_init(t_list * list) {
4      list->head = (t_list_node*)malloc(sizeof(t_list_node));
5      if (list->head == NULL) {
6          return (0);
7      }
8      list->head->next = list->head;
9      list->head->prev = list->head;
10     list->size = 0;
11     return (1);
12 }
13
14 /**
15  * Create a new linked list
16  */
17 t_list * list_new(void) {
18     t_list * list = (t_list *) malloc(sizeof(t_list));
19     if (list == NULL) {
20         return (NULL);
21     }
22
23     if (!list_init(list)) {
24         free(list);
25         return (NULL);
26     }
27
28     return (list);
29 }
30
31 /**
32  *  Add an element at the end of the list
33  */
34 void * list_add(t_list * lst, void const *content, unsigned int content_size)
35 {
36     t_list_node *node = (t_list_node*)malloc(sizeof(t_list_node) + content_size);
37     if (node == NULL) {
38         return (NULL);
39     }
40     memcpy(node + 1, content, content_size);
41
42     t_list_node *tmp = lst->head->prev;
43
44     lst->head->prev = node;
45     tmp->next = node;
46
47     node->prev = tmp;
48     node->next = lst->head;
```

```
49
50      lst->size++;
51
52      return (node + 1);
53  }
54
55  /**
56   * Add an element in head of the list
57   */
58  void * list_addfront(t_list * lst, void const *content, unsigned int content_size) {
59      t_list_node * node = (t_list_node *) malloc(sizeof(t_list_node) + content_size);
60      if (node == NULL) {
61          return (NULL);
62      }
63      memcpy(node + 1, content, content_size);
64
65      t_list_node *tmp = lst->head->next;
66
67      lst->head->next = node;
68      tmp->prev = node;
69
70      node->prev = lst->head;
71      node->next = tmp;
72
73      lst->size++;
74
75      return (node + 1);
76  }
77
78  /**
79   *  remove the given node from the list
80   */
81  void list_remove_node(t_list * lst, t_list_node *node) {
82      if (node->prev) {
83          node->prev->next = node->next;
84      }
85      if (node->next) {
86          node->next->prev = node->prev;
87      }
88
89      node->next = NULL;
90      node->prev = NULL;
91      free(node);
92      lst->size--;
93  }
94
95  /**
96   * Remove first / last element of the list. Return 1 if it was removed, 0 else
97   */
98  int list_remove_first(t_list * lst) {
99      if (lst->size == 0) {
100         return (0);
101     }
102     list_remove_node(lst, lst->head->next);
103     return (1);
104 }
105
106 int list_remove_last(t_list * lst) {
107     if (lst->size == 0) {
108         return (0);
109     }
110     list_remove_node(lst, lst->head->prev);
111     return (1);
112 }
113
114 /**
115  *  remove list head
116  */
117 void * list_pop(t_list * lst) {
118     if (lst->size == 0) {
119         return (NULL);
120     }
121
122     void * data = lst->head->next + 1;
123     if (lst->size > 0)
124     {
125         list_remove_first(lst);
126     }
127     return (data);
128 }
129
130 /** return content at the begining of the list */
131 void * list_head(t_list * lst) {
132     if (lst->size > 0) {
133         return ((void*)lst->head->next + 1);
134     }
135     return (NULL);
```

```
136 }
137
138
139 /** remove if the comparison return elements are equals (works like strcmp) */
140 int list_remove(t_list * lst, t_cmp_function cmpf, void * cmpd) {
141     t_list_node *node;
142
143     node = lst->head->next;
144     while (node != lst->head) {
145         if (cmpf(node + 1, cmpd) == 0) {
146             list_remove_node(lst, node);
147             return (1);
148         }
149         node = node->next;
150     }
151     return (0);
152 }
153
154 /**
155  *  Return the list node data which match with the given comparison function
156  *  and reference data. (cmpf should acts like 'strcmp()')
157  */
158 void * list_get(t_list * lst, t_cmp_function cmpf, void * cmpd) {
159     if (lst->size == 0) {
160         return (NULL);
161     }
162
163     if (cmpf(lst->head + 1, cmpf) == 0) {
164         return (lst->head);
165     }
166
167     t_list_node *node = lst->head->next;
168     while (node != lst->head) {
169         if (cmpf(node + 1, cmpd) == 0) {
170             return (node + 1);
171         }
172         node = node->next;
173     }
174
175     return (NULL);
176 }
177
178 /**
179  * Remove the node which datas match with the given comparison function
180  * and the given data reference
181  */
182 void list_delete(t_list * lst) {
183     if (lst->size == 0) {
184         goto end;
185     }
186
187     list_clear(lst);
188
189 end:
190     lst->head = NULL;
191     lst->size = 0;
192 }
193
194 /**
195  *  clear the list : remove every nodes
196  */
197 void list_clear(t_list * lst) {
198
199     t_list_node * node = lst->head->next;
200     while (node != lst->head) {
201         t_list_node *next = node->next;
202         free(node);
203         node = next;
204     }
205
206     free(lst->head);
207     list_init(lst);
208 }
209
210 /**
211  * Return a buffer which holds pointers to every elements of the list, allocated with 'malloc()'
212  */
213 void * list_buffer(t_list * lst) {
214     void ** buffer = (void**)malloc(sizeof(void*) * (lst->size + 1));
215     if (buffer == NULL) {
216         return (NULL);
217     }
218
219     t_list_node *node = lst->head->next;
220     unsigned int i = 0;
221
222     while (node != lst->head) {
```

```
223         buffer[i] = (void*)(node + 1);
224         ++i;
225         node = node->next;
226     }
227
228     buffer[i] = NULL;
229     return ((void*)buffer);
230 }
231
232 /**
233  *  iterate the function to every node content of the list
234  */
235 void list_iterate(t_list * lst, t_function f)
236 {
237     LIST_ITER_START(lst, void * , content) {
238         f(content);
239     }
240     LIST_ITER_END(lst, void * , content)
241 }
```

4 Feuille de calcul Python

python/lagrange.py

```python
import numpy as np
import matplotlib.pyplot as plt
import math

n = 1000
a = -1
b =  1
h = (b - a) / float(n)

x   = [a + i * h for i in range(0, n)]
dy  = [math.acos(xi) - ((-0.69813170079773212 * xi * xi - 0.87266462599716477) * xi +
    1.5707963267948966) for xi in x]
y1  = [math.acos(xi) for xi in x]
y2  = [(-0.69813170079773212 * xi * xi - 0.87266462599716477) * xi + 1.5707963267948966 for xi in
     x]

plt.plot(x, dy, label="dy")
plt.plot(x, y1, label="acos")
plt.plot(x, y2, label="acos Lagrange")

plt.show()
```

python/memory.py

```python
import numpy as np
import matplotlib.pyplot as plt

Ko = 1 / 1024.0;
Mo = 1 / (1024.0 * 1024.0);
nmin = 4
nmax = 32 # nombre de point a calculer
SUB = 16.0 # nombre de points par subdivision de terrain

# nombre de points total sur la carte
def N(n):
    return (n * n)

# nombre de triangle total sur la carte
def T(n):
    return (2 * (n - 1) * (n - 1))

# nombre d'indice pour relie les triangles
def I(n):
    return (3 * T(n))

def M1(n):
    return (I(n) * 28)

def M2(n):
    return (N(n) * 28 + 2 * I(n))

def M3(n):
    return (N(n) * 16 + n / SUB * (16 * 4 + 2 * 4) + I(SUB))

x       = [n for n in range(nmin, nmax)]
yM1  = [M1(n) * Mo for n in x]
yM2  = [M2(n) * Mo for n in x]
yM3  = [M3(n) * Mo for n in x]

rM1 = [M1(n) / T(n) for n in x]
rM2  = [M2(n) / T(n) for n in x]
rM3  = [M3(n) / T(n) for n in x]

#plt.plot(x, yM1, label="M1(n)")
#plt.plot(x, yM2, label="M2(n)")
#plt.plot(x, yM3, label="M3(n)")

plt.plot(x, rM1, label="M1(n) / T(n)")
plt.plot(x, rM2, label="M2(n) / T(n)")
plt.plot(x, rM3, label="M3(n) / T(n)")

plt.legend()
plt.show()
```