CS/SE 4F03 Final Project
Rendering 3D Fractals on a GPU

Stephen Wynn-Williams
Dominik Kaukinen
Logan Crump
Rob Epp

April 11, 2016

Description of Parallelization

The code can be paralleled in two distinct sections. The first is across the frame and the second is across the path. Parallelization across the frame takes place in renderer.cc in the nested for loops that iterate over each pixel of the frame. Since the pixels are all data independent of each other this was an ideal place to break off into multiple workers. We used `acc parallel loop independent private(color, to, pix_data, farPoint)` around the outer loop and `acc loop private(color, to, pix_data, farPoint)` around the inner loop. Since there are calls to `local_UnProject`, `rayMarch` and `getColour` we needed to add `acc routine seq` to their definitions and all the subsequent functions that they called. To improve performance we added a `data` pragma around the outer loop to reduce the amount of copying between the host and the device.

Our program has the ability to generate frames automatically, but a manual path file can also be used. Each line in the path file contains a camera position, look at vector, up vector and field of view. If a path file is given as an input argument, our program will create an array of camera parameters and generate a video frame for every camera parameter given. Since each frame is also data independent of each other we can scatter the array across MPI nodes to compute sections of the path in parallel.

The automatic path planner determines the next frame based on the current frame so we do not currently have the ability to separate it between MPI nodes. There is a potential to return the path that the automated path planner used and then if the video needed to be rendered again at higher detail or resolution it could be used as a manual path.

Frame Generating Algorithm

MandelBox vs MandelBulb Performance

Performance using params_bulb.dat. CPU time averaged over 10 frames and GPU time averaged over 7200 frames. Times include automated path planning and .bmp writes to a solid state hard drive after every frame. There is a consistent 20% performance penalty from using a mechanical spinning platter hard drive regardless of whether a CPU or GPU is used to generate frames. The time to encode the frames into a video is not included in any of the tables.

CPU	AMD FX-9590 Vishera 8-Core 4.7 GHz
RAM	16GB DDR3 1600MHz
GPU	Nvidia GeForce GTX TITAN X

Table 1: Dominik's Desktop Specifications

	1920x1080		3840x2160	
Cores	Time/Frame Av. (s)	Speed Up	Time/Frame Av. (s)	Speed Up
1	30.50	-	124.50	-
2	15.44	1.95x	61.71	2.02x
4	8.31	3.61x	33.72	3.69x
8	4.92	6.11x	19.64	6.34x
Titan X	0.11	272.73	0.36	341.76x

Table 2: Mandlebulb Performance and Speedup using Dominick's Desktop (Table: 1)

Reproduction of Results

Our results can be obtained by typing `$ make acc` and executing `$ ./mandelbulb params_bulb.dat` to generate the frames. The path is determined by an automatic path planner. The file auto_path.dat must be in the root directory, it is used as the starting point for the automated path planner. To create a video from the frames type `$ make video`. The video will be created in the `output` folder of the root directory named `output.mp4`.

auto_path.dat

```
0.8 0.11 0.7 0 0 0 0 1 0 1.3
```

params_bulb.dat

```
# IMAGE
# width height
#3840 2160
1920 1080
# detail level, the smaller the more detailed (-3)
-3
# MANDELBOX
# n/a, rMin (bailout), rFixed (power) (0 4.0 8)
0  4.0 14
# max number of iterations, escape time
100 0
# COLORING
# type 0 or 1
1
# brightness
1.2
# IMAGE FILE NAME
image
```

Source Code

Note that all code has been converted to single precision floating point arithmetic, uses MPI, OpenMP, openACC and has completely autonomous path planning.

3d.cc

```
#include <stdio.h>
#include <stdlib.h>
```

```
#include <string.h>
#ifdef _OPENACC
#include <accelmath.h>
#else
#include <math.h>
#endif
#include "3d.h"

# define M_PI   3.14159265358979323846  /* pi */

//-------------------------------------------------------------------------------------------
//when projection and modelview matricies are static (computed only once, and camera does not mover)
#pragma acc routine seq
int UnProject(float winX, float winY, CameraParams camP, float *obj)
{
    //Transformation vectors
    float in[4], out[4];

    //Transformation of normalized coordinates between -1 and 1
    in[0]=(winX-(float)(camP.viewport[0]))/(float)(camP.viewport[2])*2.0-1.0;
    in[1]=(winY-(float)(camP.viewport[1]))/(float)(camP.viewport[3])*2.0-1.0;
    in[2]=2.0-1.0;
    in[3]=1.0;

    //Objects coordinates
    float *matrix = camP.matInvProjModel;
    MultiplyMatrixByVector(out, matrix, in);

    if(out[3]==0.0)
        return 0;

    out[3] = 1.0/out[3];
    obj[0] = out[0]*out[3];
    obj[1] = out[1]*out[3];
    obj[2] = out[2]*out[3];
    return 1;
}


void LoadIdentity(float *matrix){
    matrix[0] = 1.0;
    matrix[1] = 0.0;
    matrix[2] = 0.0;
    matrix[3] = 0.0;

    matrix[4] = 0.0;
    matrix[5] = 1.0;
    matrix[6] = 0.0;
    matrix[7] = 0.0;

    matrix[8] = 0.0;
    matrix[9] = 0.0;
    matrix[10] = 1.0;
    matrix[11] = 0.0;

    matrix[12] = 0.0;
    matrix[13] = 0.0;
    matrix[14] = 0.0;
    matrix[15] = 1.0;
}

//-------------------------------------------------------------------------------------------
void Perspective(float fov, float aspect, float zNear, float zFar, float *projMat)
{
    float ymax, xmax;

    ymax = zNear * tan(fov * M_PI / 360.0);
    //ymin = -ymax;
```

```
    //xmin = -ymax * aspectRatio;
    xmax = ymax * aspect;
    Frustum(-xmax, xmax, -ymax, ymax, zNear, zFar, projMat);
}

void Frustum(float left, float right, float bottom, float top, float znear, float zfar, float *matrix)
{
    float temp, temp2, temp3, temp4;
    temp = 2.0 * znear;
    temp2 = right - left;
    temp3 = top - bottom;
    temp4 = zfar - znear;
    matrix[0] = temp / temp2;
    matrix[1] = 0.0;
    matrix[2] = 0.0;
    matrix[3] = 0.0;
    matrix[4] = 0.0;
    matrix[5] = temp / temp3;
    matrix[6] = 0.0;
    matrix[7] = 0.0;
    matrix[8] = (right + left) / temp2;
    matrix[9] = (top + bottom) / temp3;
    matrix[10] = (-zfar - znear) / temp4;
    matrix[11] = -1.0;
    matrix[12] = 0.0;
    matrix[13] = 0.0;
    matrix[14] = (-temp * zfar) / temp4;
    matrix[15] = 0.0;
}
//--------------------------------------------------------------------------
void LookAt(float *eye, float *target, float *upV, float *modelMatrix)
{
    float forward[3], side[3], up[3];
    float matrix2[16], resultMatrix[16];
    //------------------
    forward[0] = target[0] - eye[0];
    forward[1] = target[1] - eye[1];
    forward[2] = target[2] - eye[2];
    NormalizeVector(forward);
    //------------------
    //Side = forward x up
    ComputeNormalOfPlane(side, forward, upV);
    NormalizeVector(side);
    //------------------
    //Recompute up as: up = side x forward
    ComputeNormalOfPlane(up, side, forward);
    //------------------
    matrix2[0] = side[0];
    matrix2[4] = side[1];
    matrix2[8] = side[2];
    matrix2[12] = 0.0;
    //------------------
    matrix2[1] = up[0];
    matrix2[5] = up[1];
    matrix2[9] = up[2];
    matrix2[13] = 0.0;
    //------------------
    matrix2[2] = -forward[0];
    matrix2[6] = -forward[1];
    matrix2[10] = -forward[2];
    matrix2[14] = 0.0;
    //------------------
    matrix2[3] = matrix2[7] = matrix2[11] = 0.0;
    matrix2[15] = 1.0;
    //------------------
    MultiplyMatrices(resultMatrix, modelMatrix, matrix2);
    Translate(resultMatrix, -eye[0], -eye[1], -eye[2]);
    //------------------
```

```
    memcpy(modelMatrix, resultMatrix, 16*sizeof(float));
}


void NormalizeVector(float *v)
{
    float m = 1.0/sqrtf(v[0]*v[0]+v[1]*v[1]+v[2]*v[2]);
    v[0] *= m;
    v[1] *= m;
    v[2] *= m;
}

void ComputeNormalOfPlane(float *normal, float *v1, float *v2)
{
    normal[0] = v1[1] * v2[2] - v1[2] * v2[1];
    normal[1] = v1[2] * v2[0] - v1[0] * v2[2];
    normal[2] = v1[0] * v2[1] - v1[1] * v2[0];
}

void MultiplyMatrices(float *result, const float *matrix1, const float *matrix2)
{
    result[0]=matrix1[0]*matrix2[0]+
        matrix1[4]*matrix2[1]+
        matrix1[8]*matrix2[2]+
        matrix1[12]*matrix2[3];
    result[1]=matrix1[1]*matrix2[0]+
        matrix1[5]*matrix2[1]+
        matrix1[9]*matrix2[2]+
        matrix1[13]*matrix2[3];
    result[2]=matrix1[2]*matrix2[0]+
        matrix1[6]*matrix2[1]+
        matrix1[10]*matrix2[2]+
        matrix1[14]*matrix2[3];
    result[3]=matrix1[3]*matrix2[0]+
        matrix1[7]*matrix2[1]+
        matrix1[11]*matrix2[2]+
        matrix1[15]*matrix2[3];
    result[4]=matrix1[0]*matrix2[4]+
        matrix1[4]*matrix2[5]+
        matrix1[8]*matrix2[6]+
        matrix1[12]*matrix2[7];
    result[5]=matrix1[1]*matrix2[4]+
        matrix1[5]*matrix2[5]+
        matrix1[9]*matrix2[6]+
        matrix1[13]*matrix2[7];
    result[6]=matrix1[2]*matrix2[4]+
        matrix1[6]*matrix2[5]+
        matrix1[10]*matrix2[6]+
        matrix1[14]*matrix2[7];
    result[7]=matrix1[3]*matrix2[4]+
        matrix1[7]*matrix2[5]+
        matrix1[11]*matrix2[6]+
        matrix1[15]*matrix2[7];
    result[8]=matrix1[0]*matrix2[8]+
        matrix1[4]*matrix2[9]+
        matrix1[8]*matrix2[10]+
        matrix1[12]*matrix2[11];
    result[9]=matrix1[1]*matrix2[8]+
        matrix1[5]*matrix2[9]+
        matrix1[9]*matrix2[10]+
        matrix1[13]*matrix2[11];
    result[10]=matrix1[2]*matrix2[8]+
        matrix1[6]*matrix2[9]+
        matrix1[10]*matrix2[10]+
        matrix1[14]*matrix2[11];
    result[11]=matrix1[3]*matrix2[8]+
        matrix1[7]*matrix2[9]+
        matrix1[11]*matrix2[10]+
```

```
        matrix1[15]*matrix2[11];
    result[12]=matrix1[0]*matrix2[12]+
        matrix1[4]*matrix2[13]+
        matrix1[8]*matrix2[14]+
        matrix1[12]*matrix2[15];
    result[13]=matrix1[1]*matrix2[12]+
        matrix1[5]*matrix2[13]+
        matrix1[9]*matrix2[14]+
        matrix1[13]*matrix2[15];
    result[14]=matrix1[2]*matrix2[12]+
        matrix1[6]*matrix2[13]+
        matrix1[10]*matrix2[14]+
        matrix1[14]*matrix2[15];
    result[15]=matrix1[3]*matrix2[12]+
        matrix1[7]*matrix2[13]+
        matrix1[11]*matrix2[14]+
        matrix1[15]*matrix2[15];
}
#pragma acc routine seq
void MultiplyMatrixByVector(float *resultvector, float *matrix, float *pvector)
{
    resultvector[0]=matrix[0]*pvector[0]+matrix[4]*pvector[1]+matrix[8]*pvector[2]+matrix[12]*pvector[3];
    resultvector[1]=matrix[1]*pvector[0]+matrix[5]*pvector[1]+matrix[9]*pvector[2]+matrix[13]*pvector[3];
    resultvector[2]=matrix[2]*pvector[0]+matrix[6]*pvector[1]+matrix[10]*pvector[2]+matrix[14]*pvector[3];
    resultvector[3]=matrix[3]*pvector[0]+matrix[7]*pvector[1]+matrix[11]*pvector[2]+matrix[15]*pvector[3];
}

#define SWAP_ROWS(a, b) { float *_tmp = a; (a)=(b); (b)=_tmp; }
#define MAT(m,r,c) (m)[(c)*4+(r)]

int InvertMatrix(float *m, float *out){
    float wtmp[4][8];
    float m0, m1, m2, m3, s;
    float *r0, *r1, *r2, *r3;
    r0 = wtmp[0], r1 = wtmp[1], r2 = wtmp[2], r3 = wtmp[3];
    r0[0] = MAT(m, 0, 0), r0[1] = MAT(m, 0, 1),
        r0[2] = MAT(m, 0, 2), r0[3] = MAT(m, 0, 3),
        r0[4] = 1.0, r0[5] = r0[6] = r0[7] = 0.0,
        r1[0] = MAT(m, 1, 0), r1[1] = MAT(m, 1, 1),
        r1[2] = MAT(m, 1, 2), r1[3] = MAT(m, 1, 3),
        r1[5] = 1.0, r1[4] = r1[6] = r1[7] = 0.0,
        r2[0] = MAT(m, 2, 0), r2[1] = MAT(m, 2, 1),
        r2[2] = MAT(m, 2, 2), r2[3] = MAT(m, 2, 3),
        r2[6] = 1.0, r2[4] = r2[5] = r2[7] = 0.0,
        r3[0] = MAT(m, 3, 0), r3[1] = MAT(m, 3, 1),
        r3[2] = MAT(m, 3, 2), r3[3] = MAT(m, 3, 3),
        r3[7] = 1.0, r3[4] = r3[5] = r3[6] = 0.0;
    /* choose pivot - or die */
    if (fabsf(r3[0]) > fabsf(r2[0]))
        SWAP_ROWS(r3, r2);
    if (fabsf(r2[0]) > fabsf(r1[0]))
        SWAP_ROWS(r2, r1);
    if (fabsf(r1[0]) > fabsf(r0[0]))
        SWAP_ROWS(r1, r0);
    if (0.0 == r0[0])
        return 0;
    /* eliminate first variable     */
    m1 = r1[0] / r0[0];
    m2 = r2[0] / r0[0];
    m3 = r3[0] / r0[0];
    s = r0[1];
    r1[1] -= m1 * s;
    r2[1] -= m2 * s;
    r3[1] -= m3 * s;
    s = r0[2];
    r1[2] -= m1 * s;
    r2[2] -= m2 * s;
    r3[2] -= m3 * s;
```

```
s = r0[3];
r1[3] -= m1 * s;
r2[3] -= m2 * s;
r3[3] -= m3 * s;
s = r0[4];
if (s != 0.0) {
    r1[4] -= m1 * s;
    r2[4] -= m2 * s;
    r3[4] -= m3 * s;
}
s = r0[5];
if (s != 0.0) {
    r1[5] -= m1 * s;
    r2[5] -= m2 * s;
    r3[5] -= m3 * s;
}
s = r0[6];
if (s != 0.0) {
    r1[6] -= m1 * s;
    r2[6] -= m2 * s;
    r3[6] -= m3 * s;
}
s = r0[7];
if (s != 0.0) {
    r1[7] -= m1 * s;
    r2[7] -= m2 * s;
    r3[7] -= m3 * s;
}
/* choose pivot - or die */
if (fabsf(r3[1]) > fabsf(r2[1]))
    SWAP_ROWS(r3, r2);
if (fabsf(r2[1]) > fabsf(r1[1]))
    SWAP_ROWS(r2, r1);
if (0.0 == r1[1])
    return 0;
/* eliminate second variable */
m2 = r2[1] / r1[1];
m3 = r3[1] / r1[1];
r2[2] -= m2 * r1[2];
r3[2] -= m3 * r1[2];
r2[3] -= m2 * r1[3];
r3[3] -= m3 * r1[3];
s = r1[4];
if (0.0 != s) {
    r2[4] -= m2 * s;
    r3[4] -= m3 * s;
}
s = r1[5];
if (0.0 != s) {
    r2[5] -= m2 * s;
    r3[5] -= m3 * s;
}
s = r1[6];
if (0.0 != s) {
    r2[6] -= m2 * s;
    r3[6] -= m3 * s;
}
s = r1[7];
if (0.0 != s) {
    r2[7] -= m2 * s;
    r3[7] -= m3 * s;
}
/* choose pivot - or die */
if (fabsf(r3[2]) > fabsf(r2[2]))
    SWAP_ROWS(r3, r2);
if (0.0 == r2[2])
    return 0;
/* eliminate third variable */
```

```
    m3 = r3[2] / r2[2];
    r3[3] -= m3 * r2[3], r3[4] -= m3 * r2[4],
        r3[5] -= m3 * r2[5], r3[6] -= m3 * r2[6], r3[7] -= m3 * r2[7];
    /* last check */
    if (0.0 == r3[3])
        return 0;
    s = 1.0 / r3[3];    /* now back substitute row 3 */
    r3[4] *= s;
    r3[5] *= s;
    r3[6] *= s;
    r3[7] *= s;
    m2 = r2[3];     /* now back substitute row 2 */
    s = 1.0 / r2[2];
    r2[4] = s * (r2[4] - r3[4] * m2), r2[5] = s * (r2[5] - r3[5] * m2),
        r2[6] = s * (r2[6] - r3[6] * m2), r2[7] = s * (r2[7] - r3[7] * m2);
    m1 = r1[3];
    r1[4] -= r3[4] * m1, r1[5] -= r3[5] * m1,
        r1[6] -= r3[6] * m1, r1[7] -= r3[7] * m1;
    m0 = r0[3];
    r0[4] -= r3[4] * m0, r0[5] -= r3[5] * m0,
        r0[6] -= r3[6] * m0, r0[7] -= r3[7] * m0;
    m1 = r1[2];     /* now back substitute row 1 */
    s = 1.0 / r1[1];
    r1[4] = s * (r1[4] - r2[4] * m1), r1[5] = s * (r1[5] - r2[5] * m1),
        r1[6] = s * (r1[6] - r2[6] * m1), r1[7] = s * (r1[7] - r2[7] * m1);
    m0 = r0[2];
    r0[4] -= r2[4] * m0, r0[5] -= r2[5] * m0,
        r0[6] -= r2[6] * m0, r0[7] -= r2[7] * m0;
    m0 = r0[1];     /* now back substitute row 0 */
    s = 1.0 / r0[0];
    r0[4] = s * (r0[4] - r1[4] * m0), r0[5] = s * (r0[5] - r1[5] * m0),
        r0[6] = s * (r0[6] - r1[6] * m0), r0[7] = s * (r0[7] - r1[7] * m0);
    MAT(out, 0, 0) = r0[4];
    MAT(out, 0, 1) = r0[5], MAT(out, 0, 2) = r0[6];
    MAT(out, 0, 3) = r0[7], MAT(out, 1, 0) = r1[4];
    MAT(out, 1, 1) = r1[5], MAT(out, 1, 2) = r1[6];
    MAT(out, 1, 3) = r1[7], MAT(out, 2, 0) = r2[4];
    MAT(out, 2, 1) = r2[5], MAT(out, 2, 2) = r2[6];
    MAT(out, 2, 3) = r2[7], MAT(out, 3, 0) = r3[4];
    MAT(out, 3, 1) = r3[5], MAT(out, 3, 2) = r3[6];
    MAT(out, 3, 3) = r3[7];
    return 1;
}

void Translate(float *result, float x, float y, float z){
    float matrix[16], resultMatrix[16];

    LoadIdentity(matrix);
    matrix[12] = x;
    matrix[13] = y;
    matrix[14] = z;

    MultiplyMatrices(resultMatrix, result, matrix);
    memcpy(result, resultMatrix, 16*sizeof(float));
}
```

3d.h

```
#ifndef _3d_H
#define _3d_H

#define NEAR 1
#define FAR 100

#include "camera.h"
#include "renderer.h"
```

```
void   LoadIdentity   (float *matrix);
void   Perspective    (float fov, float aspect, float zNear, float zFar, float *projMatrix);
void   Frustum        (float left, float right, float bottom, float top, float znear, float zfar, float *matrix);
void   LookAt         (float *eye, float *target, float *up, float *modelMatrix);
float  LengthVector   (float *vector);
void   NormalizeVector(float *vector);
void   ComputeNormalOfPlane(float *normal, float *v1, float *v2);
void   MultiplyMatrices(float *result, const float *matrix1, const float *matrix2);
void   MultiplyMatrixByVector(float *resultvector, float *matrix, float *pvector);
int    InvertMatrix(float *m, float *out);
void   Translate(float *result, float x, float y, float z);

#endif
```

camera.h

```
#ifndef _CAMERA_H
#define _CAMERA_H

typedef struct
{
    float camPos[3];
    float camTarget[3];
    float camUp[3];
    float fov;
    float matModelView[16];
    float matProjection[16];
    float matInvProjModel[16];
    int    viewport[4];
} CameraParams;

#endif
```

color.h

unmodified

distance_est.cc

```
#include "vector3d.h"
#include "mandelbox.h"

extern MandelBoxParams mandelBox_params;
extern float MandelBoxDE(const vec3 &pos, const MandelBoxParams &mPar, float c1, float c2);
extern float MandelBulbDistanceEstimator(const vec3 &p0, MandelBoxParams &params);

//Distance Estimator Field Selector
float DE(vec3 &p)
{
  //double c1 = fabs(mandelBox_params.scale - 1.0);
  //double c2 = pow( fabs(mandelBox_params.scale), 1 - mandelBox_params.num_iter);
  //double d = MandelBoxDE(p, mandelBox_params, c1, c2);
  //return d;

  return MandelBulbDistanceEstimator(p, mandelBox_params);
}
```

getcolor.cc

```
#include "color.h"
#include "renderer.h"
#include "vector3d.h"
#include <math.h>
#include <algorithm>
```

```
using namespace std;

//---lightning and colouring---------
#define CAM_LIGHT {.x=1.0,.y=1.0,.z=1.0}
#define CAM_LIGHT_W 1.8 // 1.27536;
#define CAM_LIGHT_MIN 0.3 // 0.48193;
//-----------------------------------
#define BASE_COLOR {.x=1.0, .y=1.0, .z=1.0}
#define BACK_COLOR {.x=0.4,.y=0.4,.z=0.4}
//-----------------------------------

#pragma acc routine seq
void lighting(const vec3 &n, const vec3 &color, const vec3 &pos, const vec3 &direction,  vec3 &outV)
{
    vec3 CamLight = CAM_LIGHT;
    float CamLightW = CAM_LIGHT_W;
    float CamLightMin = CAM_LIGHT_MIN;

    vec3 nn;

    SUBTRACT_POINT_FLOAT(nn, n, 1.0);
    float d = 0.0;
    DOT(d, direction, nn);
    float ambient = MAX(CamLightMin, d) * CamLightW;
    MULT_FLOAT(outV, CamLight, ambient);
    MULT_VEC(outV, outV, color);
}

#pragma acc routine seq
void getColour(const pixelData &pixData, const RenderParams &render_params,
                const vec3 &from, const vec3  &direction, vec3  &hitColor)
{
    vec3 baseColor = BASE_COLOR;
    vec3 backColor = BACK_COLOR;

    //colouring and lightning
    hitColor = baseColor;

    if (pixData.escaped == false)
    {
        //apply lighting
        lighting(pixData.normal, hitColor, pixData.hit, direction, hitColor);

        //add normal based colouring
        if(render_params.colourType == 0 || render_params.colourType == 1)
        {
            MULT_VEC(hitColor, hitColor, pixData.normal);
            ADD_FLOAT(hitColor, hitColor, 1.0);
            MULT_FLOAT(hitColor, hitColor, 0.5);
            MULT_FLOAT(hitColor, hitColor, render_params.brightness);

            //gamma correction
            CLAMP(hitColor, 0.0, 1.0);
            MULT_VEC(hitColor, hitColor, hitColor);
        }
        if(render_params.colourType == 1)
        {
            //"swap" colors
            float t = hitColor.x;
            hitColor.x = hitColor.z;
            hitColor.z = t;
        }
    }
    else
        //we have the background colour
        hitColor = backColor;

    //return hitColor;
```

```
}
```

getnextframe.c

```
#include <stdio.h>
#include <stdlib.h>
// #include <string.h>

#ifdef _OPENACC
#include <accelmath.h>
#else
#include <math.h>
#endif
#include "vector3d.h"
#include "renderer.h"
#include "camera.h"

#define STEP_SIZE 0.001

#define STEP_EPSILON 1.1052
#define STEP_THETA 0.001
#define PI 3.141592654

inline void moveVec(CameraParams * next_frame, float max_dist);
inline void rotateVec(CameraParams * next_frame, float xTheta, float yTheta);

void getNextFrame(CameraParams * next_frame, RenderParams * renderer_params, float * dist_matrix)
{
    // printf("direction of largest distance x = %f, y = %f\n",
    // next_frame->camTarget[0],
    // next_frame->camPos[0]);
    int i,j;
    vec3 new_target;
    //buffer distance to walls
    float max_dist = 0.1;
    for (i = 0 ; i < renderer_params->height ; i++)
    {
        for(j = 0 ; j < renderer_params->width ; j++)
        {
            if(max_dist < dist_matrix[(4 * i * renderer_params->width) + (4 * j)])
            {
                max_dist = dist_matrix[(4 * i * renderer_params->width) + (4 * j)];
                new_target.x = dist_matrix[(4 * i * renderer_params->width) + (4 * j) + 1];
                new_target.y = dist_matrix[(4 * i * renderer_params->width) + (4 * j) + 2];
                new_target.z = dist_matrix[(4 * i * renderer_params->width) + (4 * j) + 3];
            }

        }
    }

    next_frame->camTarget[0] *= 0.98;
    next_frame->camTarget[1] *= 0.98;
    next_frame->camTarget[2] *= 0.98;

    MULT_FLOAT(new_target, new_target, 0.02);

    next_frame->camTarget[0] += new_target.x;
    next_frame->camTarget[1] += new_target.y;
    next_frame->camTarget[2] += new_target.z;

    if(max_dist > 10 * STEP_SIZE){
        moveVec(next_frame, max_dist);
    }
}

/*
Determine direction of the cam pos
```

```
rotation about x:

    |1     0         0| |x|   |        x      |   |x'|
    |0   cos      sin | |y| = |y cos   z sin  | = |y'|
    |0   sin      cos | |z|   |y sin  + z cos |   |z'|

rotation about y:

    | cos     0   sin | |x|   | x cos  + z sin |   |x'|
    |  0      1      0| |y| = |         y      | = |y'|
    | sin     0   cos | |z|   | x sin  + z cos |   |z'|

*/
inline void rotateVec(CameraParams * next_frame, float xTheta, float yTheta){
    float x = next_frame-> camTarget[0] - next_frame-> camPos[0];
    float y = next_frame-> camTarget[1] - next_frame-> camPos[1];
    float z = next_frame-> camTarget[2] - next_frame-> camPos[2];

    //first rotate in x
    float newY = y * cosf(xTheta) - z * sinf(xTheta);
    float newZ = y * sinf(xTheta) + z * cosf(xTheta);

    //then rotate in y
    float newX = x * cosf(yTheta) - z * sinf(yTheta);
    newZ = z * cosf(yTheta) - x * sinf(yTheta);
    // printf("y = %f, newy = %f\n", y, newY);
    next_frame-> camTarget[0] = newX;
    next_frame-> camTarget[1] = newY;
    next_frame-> camTarget[2] = newZ;
}

//move the cam pos one unit in the direction of the cam target

inline void moveVec(CameraParams * next_frame, float max_dist){
    float x = next_frame-> camTarget[0] - next_frame-> camPos[0];
    float y = next_frame-> camTarget[1] - next_frame-> camPos[1];
    float z = next_frame-> camTarget[2] - next_frame-> camPos[2];

    // next_frame-> camPos[0] = next_frame-> camPos[0] + ((x > 0) - (x < 0))*;
    // next_frame-> camPos[1] = next_frame-> camPos[1] + ((y > 0) - (y < 0))*;
    // next_frame-> camPos[2] = next_frame-> camPos[2] + ((z > 0) - (z < 0))*(max_dist);

    float mag = sqrtf((x * x) + (y * y) + (z * z));
    mag *= (1 / STEP_SIZE);

    x = x / mag;
    y = y / mag;
    z = z / mag;

    next_frame-> camTarget[0] += x;
    next_frame-> camTarget[1] += y;
    next_frame-> camTarget[2] += z;

    next_frame-> camPos[0] += x;
    next_frame-> camPos[1] += y;
    next_frame-> camPos[2] += z;
}
```

getparams.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "renderer.h"
#include "mandelbox.h"
#include "camera.h"
```

```
#define BUF_SIZE 1024

static char buf[BUF_SIZE];

void getParameters(char *filename, RenderParams *renP, MandelBoxParams *boxP)
{
    FILE *fp;
    int ret;

    renP->fractalType = 0;
    renP->maxRaySteps = 8000;
    renP->maxDistance = 1000;

    fp = fopen(filename,"r");

    if( !fp )
        {
            printf(" *** File %s does not exist\n", filename);
            exit(1);
        }

    int count = 0;

    while (1){
        memset(buf, 0, BUF_SIZE);

        ret = fscanf(fp, "%1023[^\n]\n", buf);
        if (ret == EOF) break;

        if(buf[0] == '#') // comment line
        continue;

        switch(count){
            //IMAGE
            //width, height
            case 0:
                sscanf(buf, "%d %d", &renP->width, &renP->height);
                break;
                //detail
            case 1:
                sscanf(buf, "%f", &renP->detail);
                break;

            //FRACTAL
            case 2:
                sscanf(buf, "%f %f %f", &boxP->scale, &boxP->rMin, &boxP->rFixed);
                break;

            case 3:
                sscanf(buf, "%d %f ", &boxP->num_iter, &boxP->escape_time);
                break;

            //COLORING
            case 4:
                sscanf(buf, "%d", &renP->colourType);
                break;
            case 5:
                sscanf(buf, "%f ", &renP->brightness);
                break;
            }
        count++;
    }
    fclose(fp);
}
```

getpath.c

```
#include <assert.h>
#include <stdio.h>
#include "camera.h"

void getPath(char *filename, CameraParams *path, int *len)
{
    FILE* fid;

    fid = fopen(filename,"r");
    assert(fid != NULL);

    int line = 0;
    int ret;

    // read as many line as possible
    while (!feof(fid))
    {
        CameraParams *camP;
        camP = &path[line];

        //parse each line for camera position, target, up vector and field of view
        ret = fscanf(fid, "%f %f %f\t%f %f %f\t%f %f %f\t%f\n",
        &camP->camPos[0],    &camP->camPos[1],    &camP->camPos[2],
        &camP->camTarget[0], &camP->camTarget[1], &camP->camTarget[2],
        &camP->camUp[0],     &camP->camUp[1],     &camP->camUp[2],
        &camP->fov);

        if(ret != 10) printf("Bad parse in pathfile, line:%d \n", line);

        line++;;
    }

    fclose(fid);
    *len = line;
}
```

init3D.c

```
#include "camera.h"
#include "renderer.h"
#include "3d.h"

void init3D(CameraParams *camP, const RenderParams *renP)
{
    //set up the viewport for the image
    camP->viewport[0] = 0;
    camP->viewport[1] = 0;
    camP->viewport[2] = renP->width;
    camP->viewport[3] = renP->height;

    //init the matricies
    LoadIdentity(camP->matModelView);
    LoadIdentity(camP->matProjection);

    //setting up camera lense
    Perspective((65*camP->fov), ((float)renP->width)/((float)renP->height), NEAR, FAR, camP->matProjection);

    //setting up model view matrix
    LookAt(camP->camPos, camP->camTarget, camP->camUp, camP->matModelView);

    //setting up the inverse(projection x model) matrix
    float temp[16];
    MultiplyMatrices(temp, camP->matProjection, camP->matModelView);
    //Now compute the inverse of matrix A
    InvertMatrix(temp, camP->matInvProjModel);
```

```
}
```

main.cc

```
#include <stdlib.h>
#include <stdio.h>
#include <assert.h>
#include "camera.h"
#include "renderer.h"
#include "mandelbox.h"
#include <string.h>
#include <sys/types.h>
#include <time.h>
#include <ctime>
#include <sys/stat.h>
#include <unistd.h>

#define MAX_FRAMES  7200
#define AUTO_FRAMES 1

void getParameters(char *filename, RenderParams *renderer_params, MandelBoxParams *mandelBox_paramsP);
void getPath      (char *filename, CameraParams *camera_path, int *len);
void getNextFrame (CameraParams * next_frame, RenderParams * renderer_params, float * dist_matrix);
void init3D       (CameraParams *camera_params, const RenderParams *renderer_params);
void renderFractal(const CameraParams &camera_params, const RenderParams &renderer_params, unsigned char* image, float * dist_ma
void saveBMP      (const char* filename, const unsigned char* image, int width, int height);


MandelBoxParams mandelBox_params;

int main(int argc, char** argv)
{
    //struct timeval timerStart;
    //struct timeval timerEnd;
    double time_spent;

    #ifdef _OPENMP
        int num_threads = atoi(argv[3]);
        omp_set_num_threads(num_threads);
        printf("num threads %d\n",num_threads);
        double p_time;
        printf("openMP timer starting\n");
        p_time = omp_get_wtime();
    //#elif _OPENACC
    //    printf("openACC starting\n");
    //    gettimeofday(&timerStart, NULL);
    #else
        printf("timer starting\n");
        clock_t begin, end;
        begin = clock();
    #endif

    // adapted from:
    //http://stackoverflow.com/questions/9314586/c-faster-way-to-check-if-a-directory-exists
    struct stat s;
    int err = stat("/path/to/frames", &s);
    if (-1 == err)
    {
        mkdir("frames", 0700);
    }
    else if (S_ISDIR(s.st_mode))
    {
        /* it's a dir */
    }
    else
    {
        /* exists but is no dir */
```

```
}

assert(argc >= 2);

char* path;
CameraParams * camera_path;
CameraParams * next_frame;
RenderParams renderer_params;
int nframes;
char frame_name[256];

char auto_path;

getParameters(argv[1], &renderer_params, &mandelBox_params);

if (argc == 3)
{
    auto_path = false;

    camera_path = (CameraParams*)malloc(MAX_FRAMES*sizeof(CameraParams));
    assert(camera_path);

    path = (char*)malloc(sizeof(char) * (strlen(argv[2]) + 1));
    memcpy(path, argv[2], strlen(argv[2]) + 1);
    printf("%s\n", path);

    getPath(path, camera_path, &nframes);
}
else
{
    auto_path = true;

    next_frame = (CameraParams*)malloc(sizeof(CameraParams));

    path = (char*)malloc(sizeof(char) * (strlen("auto_start.dat") + 1));
    sprintf(path, "auto_path.dat");
    printf("%s\n", path);

    getPath(path, next_frame, &nframes);

    nframes = AUTO_FRAMES;
}

int image_size = renderer_params.width * renderer_params.height;
unsigned char *image = (unsigned char*)malloc(3*image_size*sizeof(unsigned char));
float * dist_matrix = (float *)malloc(4 * image_size * sizeof(float));

for (int i = 0; i < nframes; ++i)
{
    if (auto_path == true)
    {
        init3D(next_frame, &renderer_params);
        renderFractal(*next_frame, renderer_params, image, dist_matrix);
        getNextFrame(next_frame, &renderer_params, dist_matrix);
    }
    else
    {
        init3D(&camera_path[i], &renderer_params);
        renderFractal(camera_path[i], renderer_params, image, dist_matrix);
    }

    // printf("frame = %d\tzero dist = %f\t x = %f\ty = %f\tz = %f\n", i, dist_matrix[0], dist_matrix[1], dist_matrix[2], di

    //TODO create render directory from input
    //TODO create starting name from number from input (in case of previously generated frames)
    snprintf(frame_name, sizeof(char) * 256, "./frames/frame_%04d.bmp", i);
    saveBMP(frame_name, image, renderer_params.width, renderer_params.height);
}
```

```
    printf("done\n");

    free(image);
    free(dist_matrix);
    if (auto_path == false)
    {
        free(camera_path);
    }
    free(path);

    #ifdef _OPENMP
        p_time = omp_get_wtime()-p_time;
        printf("time is %f\n", p_time);
    //#elif _OPENACC
    //    gettimeofday(&timerEnd, NULL);
    //    double time_acc = (timerEnd.tv_sec-timerStart.tv_sec) + (timerEnd.tv_usec - timerEnd.tv_usec)*1e-6;
    //    printf("time is %f\n", time_acc);
    #else
        end = clock();
        time_spent = (double)(end - begin) / CLOCKS_PER_SEC;
        printf("time is %f\n", time_spent);
    #endif
    return 0;
}
```

makefile

```
CC=pgc++
CXX=pgc++
#FLAGS    = -O2 -Wall
#CFLAGS   = $(FLAGS)
#CXXFLAGS = $(FLAGS)
#LDFLAGS  = -lm

PROGRAM_NAME=mandelbulb

OBJS=main.o print.o timing.o savebmp.o getparams.o getpath.o getnextframe.o 3d.o getcolor.o distance_est.o \
    mandelboxde.o raymarching.o renderer.o init3D.o mandelbulb_dist_est.o

$(PROGRAM_NAME): $(OBJS)
    $(CC) -o $@ $? $(CFLAGS) $(LDFLAGS)


acc: CFLAGS=-fast -acc -Minfo=accel -ta=tesla:cc50
acc: CXXFLAGS=-fast -acc -Minfo=accel -ta=tesla:cc50
acc: LDFLAGS=-acc -ta=tesla:cc50
acc: $(OBJS)
    $(CC) $(LDFLAGS) -o$(PROGRAM_NAME) $? -lm

omp: CFLAGS= -O2 -mp
omp: CXXFLAGS= -O2 -mp
omp: LDFLAGS= -mp
omp:$(OBJS)
    $(CC) $(LDFLAGS) -o $(PROGRAM_NAME) $? -lgomp -lm

clean:
    rm *.o $(PROGRAM_NAME)


video:
    #change to frames folder and run script
    @/bin/bash -c "  \
    pushd ./frames;  \
    ./make_video.sh; \
    popd;"
```

mandelbox.h

unodified

mandelboxde.cc

```
#include <cmath>
#include <cstdio>
#include <algorithm>
#include "color.h"
#include "mandelbox.h"
#include <math.h>

#define SQR(x) ((x)*(x))

#define COMPONENT_FOLD(x) { (x) = fabsf(x) <= 1? (x) : copysignf(2,(x))-(x); }


float MandelBoxDE(const vec3 &p0, const MandelBoxParams &params, float c1, float c2)
{
    vec3 p = p0;
    float rMin2   = SQR(params.rMin);
    float rFixed2 = SQR(params.rFixed);
    float escape  = SQR(params.escape_time);
    float dfactor = 1;
    float r2      =-1;
    const float rFixed2rMin2 = rFixed2/rMin2;

    int i = 0;
    while (i< params.num_iter && r2 < escape)
    {
        COMPONENT_FOLD(p.x);
        COMPONENT_FOLD(p.y);
        COMPONENT_FOLD(p.z);

        DOT(r2, p, p);

        if (r2<rMin2)
        {
            MULT_FLOAT(p, p, rFixed2rMin2);
            dfactor *= (rFixed2rMin2);
        }
        else
        if ( r2<rFixed2)
        {
            const float t = (rFixed2/r2);
            MULT_FLOAT(p, p, rFixed2/r2);
            dfactor *= t;
        }

        dfactor = dfactor*fabsf(params.scale)+1.0;

        MULT_FLOAT(p, p, params.scale);
        ADD_POINT(p, p, p0);
        i++;
    }
    float mag = 0.0;
    MAGNITUDE(mag, p);
    return  (mag - c1) / dfactor - c2;
}
```

print.c

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

```
void printProgress( double perc, double time )
{
    static char delete_space[80];
    static char * OutputString;
    perc *= 100;

    int sec = ceil(time);
    int hr = sec/3600;
    int t = sec%3600;
    int min = t/60;
    sec = t%60;

    OutputString = (char*)"*** completed % 5.2f%s ---  cum. time = %02d:%02d:%02d   % e (s)";
    sprintf(delete_space, OutputString, perc, "%%", hr, min, sec, time);

    fprintf( stderr, delete_space);
    unsigned int i = 0;
    for ( i = 0; i < strlen(delete_space); i++)
        fputc( 8, stderr);
}
```

raymarching.cc

```
#include <assert.h>
#include <algorithm>
#include <stdio.h>

#include "color.h"
#include "renderer.h"
#include "mandelbox.h"


#pragma acc routine seq
extern float MandelBulbDistanceEstimator(const vec3 &p0, MandelBoxParams &params);

#define DistEst(p0) MandelBulbDistanceEstimator(p0, frac_params) // Note this depends on scope...

#pragma acc routine seq
void normal (const vec3 & p, vec3 & normal, MandelBoxParams &frac_params);

#pragma acc routine seq
void rayMarch(const RenderParams &render_params, const vec3 &from, const vec3  &direction, \
                           float eps, pixelData& pix_data, MandelBoxParams &frac_params, float * tot_dist)
{
    float dist = 0.0;
    float totalDist = 0.0;
    vec3 z;

    // We will adjust the minimum distance based on the current zoom
    float epsModified = 0.0;

    int steps=0;
    vec3 p, tempDir;
    do
    {
        MULT_FLOAT(tempDir, direction, totalDist);
        //printf("tempDir x = %f, y = %f, z = %f\n", tempDir.x, tempDir.y, tempDir.z);
        ADD_POINT(p, from, tempDir);
        //printf("p x = %f, y = %f, z = %f\n", p.x, p.y, p.z);

        dist = DistEst(p);

        totalDist += .98*dist;

        epsModified = totalDist;
        epsModified*=eps;
        steps++;
```

```
    }
    while (dist > epsModified && totalDist <= render_params.maxDistance && steps < render_params.maxRaySteps);

    tot_dist[1] = p.x;
    tot_dist[2] = p.y;
    tot_dist[3] = p.z;

    vec3 hitNormal;
    if (dist < epsModified)
    {
        //we didnt escape
        tot_dist[0] = totalDist;
        pix_data.escaped = false;

        // We hit something, or reached MaxRaySteps
        pix_data.hit = p;

        //figure out the normal of the surface at this point
        MULT_FLOAT(hitNormal, direction, epsModified);
        SUBTRACT_POINT(hitNormal, p, hitNormal);
        normal(hitNormal, pix_data.normal, frac_params);
    }
    else
    {
        //we have the background colour
        pix_data.escaped = true;
        tot_dist[0] = -1;
    }
}

void normal(const vec3 & p, vec3 & normal, MandelBoxParams &frac_params)
{
    vec3 z;
    // compute the normal at p
    const float sqrt_mach_eps = 3.4527e-04; //for single precision!
    float mag = 0.0;
    MAGNITUDE(mag, p);
    float eps = MAX(mag, 1.0) * sqrt_mach_eps;

    vec3 e1 = {eps, 0,   0};
    vec3 e2= { 0 , eps, 0};
    vec3 e3 = {0  , 0, eps};
    vec3 x1, x2, x3, y1, y2, y3;
    ADD_POINT(x1, p, e1);
    ADD_POINT(x2, p, e2);
    ADD_POINT(x3, p, e3);

    SUBTRACT_POINT(y1, p, e1);
    SUBTRACT_POINT(y2, p, e2);
    SUBTRACT_POINT(y3, p, e3);

    normal.x = DistEst(x1)-DistEst(y1);
    normal.y = DistEst(x2)-DistEst(y2);
    normal.z = DistEst(x3)-DistEst(y3);

    NORMALIZE(normal);
}
```

renderer.cc

```
#include <stdio.h>
#include <string.h>

#include "color.h"
#include "mandelbox.h"
#include "camera.h"
```

```
#include "vector3d.h"
#include "3d.h"

extern MandelBoxParams mandelBox_params;

#pragma acc routine seq
extern void UnProject(float winX, float winY, CameraParams camP, float *obj);

#pragma acc routine seq
void MultiplyMatrixByVector1(float *resultvector, const float *matrix, float *pvector)
{
    resultvector[0]=matrix[0]*pvector[0]+matrix[4]*pvector[1]+matrix[8]*pvector[2]+matrix[12]*pvector[3];
    resultvector[1]=matrix[1]*pvector[0]+matrix[5]*pvector[1]+matrix[9]*pvector[2]+matrix[13]*pvector[3];
    resultvector[2]=matrix[2]*pvector[0]+matrix[6]*pvector[1]+matrix[10]*pvector[2]+matrix[14]*pvector[3];
    resultvector[3]=matrix[3]*pvector[0]+matrix[7]*pvector[1]+matrix[11]*pvector[2]+matrix[15]*pvector[3];
}

#pragma acc routine seq
void local_UnProject(float winX, float winY, const int * viewport, const float * matInvProjModel, float *obj)
{
    //Transformation vectors
    float in[4], out[4];

    //Transformation of normalized coordinates between -1 and 1
    in[0]=(winX-(float)(viewport[0]))/(float)(viewport[2])*2.0-1.0;
    in[1]=(winY-(float)(viewport[1]))/(float)(viewport[3])*2.0-1.0;
    in[2]=2.0-1.0;
    in[3]=1.0;

    //Objects coordinates
    const float *matrix = matInvProjModel;
    MultiplyMatrixByVector1(out, matrix, in);

    if(out[3]==0.0)
        return;

    out[3] = 1.0/out[3];
    obj[0] = out[0]*out[3];
    obj[1] = out[1]*out[3];
    obj[2] = out[2]*out[3];
    return;
}

#pragma acc routine seq
extern void rayMarch (const RenderParams &render_params, const vec3 &from, const vec3  &to, \
                                            float eps, pixelData &pix_data, MandelBoxParams &box_params, float * tot_dist);
#pragma acc routine seq
extern void getColour(const pixelData &pixData, const RenderParams &render_params,
                    const vec3 &from, const vec3  &direction, vec3 &hitcolor);

void renderFractal(const CameraParams &camera_params, const RenderParams &renderer_params,
                                      unsigned char* image, float * dist_matrix)
{

    const float eps = powf(10.0, renderer_params.detail);
    vec3 from;

    //from.SetDoublePoint(camera_params.camPos);
    from.x = camera_params.camPos[0];
    from.y = camera_params.camPos[1];
    from.z = camera_params.camPos[2];

    const int height = renderer_params.height;
    const int width  = renderer_params.width;

    int n = height*width;

    int viewport[4];
```

```
    float matInvProjModel[16];

    memcpy(viewport, camera_params.viewport, 4*sizeof(int));
    memcpy(matInvProjModel, camera_params.matInvProjModel, 16*sizeof(float));
    int size1 = 4*sizeof(int);
    int size2 = 16*sizeof(float);

    RenderParams renderer_params1 = renderer_params;

    vec3 color, to;
    pixelData pix_data;
    float farPoint[3];
    #pragma acc data copyout(image[0:3*n], dist_matrix[0:4*n]) copyin(eps, from, renderer_params1, mandelBox_params, viewport[:s
    {
    #pragma omp parallel for collapse(2) private(color, to, pix_data, farPoint)
    #pragma acc parallel loop independent private(color, to, pix_data, farPoint) //present(image, eps, from, renderer_params1, m
    for(int j = 0; j < height; j++)
    {
        //for each column pixel in the row
        #pragma acc loop private(color, to, pix_data, farPoint)
        for(int i = 0; i < width; i++)
        {
            local_UnProject(i, j, viewport, matInvProjModel, farPoint);

            to.x = farPoint[0] - from.x;
            to.y = farPoint[1] - from.y;
            to.z = farPoint[2] - from.z;

            NORMALIZE(to);

            rayMarch(renderer_params1, from, to, eps, pix_data, mandelBox_params, &dist_matrix[(4 * j * width) + (4 * i)]);

            getColour(pix_data,renderer_params1, from, to, color);

            int k = (j * width + i)*3;
            image[k+2] = (unsigned char)(color.x * 255);
            image[k+1] = (unsigned char)(color.y * 255);
            image[k]   = (unsigned char)(color.z * 255);
        }
    }
    }
}
```

renderer.h

```
#ifndef _RENMandelBoxDERER_H
#define _RENMandelBoxDERER_H

typedef struct
{
    int fractalType;
    int colourType;
    int super_sampling;
    float brightness;
    int width;
    int height;
    float detail;
    int maxRaySteps;
    float maxDistance;
} RenderParams;

#endif
```

savebmp.c

unmodified

timing.c

unmodified

vector3d.h

```
#ifndef vec3_h
#define vec3_h

//#include <cmath>

#ifdef _OPENACC
#include <accelmath.h>
#else
#include <math.h>
#endif

typedef struct
{
  float x, y, z;
}  vec3;

#define SET_POINT(p,v) { p.x=v.x; p.y=v.y; p.z=v.z; }

#define SUBTRACT_POINT(p,v,u)     \
  {                               \
  p.x=(v.x)-(u.x);                \
  p.y=(v.y)-(u.y);                \
  p.z=(v.z)-(u.z);                \
}

#define ADD_POINT(p,v,u)          \
  {                               \
  p.x=(v.x)+(u.x);                \
  p.y=(v.y)+(u.y);                \
  p.z=(v.z)+(u.z);                \
}

#define SUBTRACT_POINT_DOUBLE(p,v,u)\
{                                 \
  p.x=(v.x)-(u);                  \
  p.y=(v.y)-(u);                  \
  p.z=(v.z)-(u);                  \
}

#define SUBTRACT_POINT_FLOAT(p,v,u) \
{                                     \
  SUBTRACT_POINT_DOUBLE(p, v, u)    \
}

#define ADD_DOUBLE(p,v,d)\
{                          \
  p.x=(v.x) + d;           \
  p.y=(v.y) + d;           \
  p.z=(v.z) + d;           \
}

#define ADD_FLOAT(p,v,d) \
{                        \
  ADD_DOUBLE(p, v, d)    \
}

#define MULT_DOUBLE(p, v, d)\
{\
  p.x=(v.x) * d;\
  p.y=(v.y) * d;\
  p.z=(v.z) * d;\
}
```

```
#define MULT_FLOAT(p, v, d)  \
{                            \
  MULT_DOUBLE(p, v, d)       \
}

#define MULT_VEC(p , v, u)    \
{                             \
  p.x=(v.x)*(u.x);            \
  p.y=(v.y)*(u.y);            \
  p.z=(v.z)*(u.z);            \
}

#define NORMALIZE(p) {                                \
    float fMag = ( p.x*p.x + p.y*p.y + p.z*p.z );     \
    if (fMag != 0)                                    \
      {                                               \
    float fMult = 1.0/sqrtf(fMag);                    \
    p.x *= fMult;                                     \
    p.y *= fMult;                                     \
    p.z *= fMult;                                     \
      }                                               \
  }

#define CLAMP(v, min, max) \
{\
  v.x = v.x<min?min:v.x;\
  v.x = v.x>max?max:v.x;\
  v.y = v.y<min?min:v.y;\
  v.y = v.y>max?max:v.y;\
  v.z = v.z<min?min:v.z;\
  v.z = v.z>max?max:v.z;\
}

#define MAGNITUDE(m,p)     ({ m=sqrtf( p.x*p.x + p.y*p.y + p.z*p.z ); })

#define DOT(d,p, v) { d=( p.x*v.x + p.y*v.y + p.z*v.z ); }

#define MAX(a,b) ( ((a)>(b))? (a):(b))

#define VEC(v,a,b,c) { v.x = a; v.y = b; v.z = c; }

#endif
```