CS 4F03 – Distributed Systems:

Final Project Report

Stuart Douglas – 1214422
Matthew Pagnan – 1208693

April 4, 2016

Contents

1 Description of Parallelization

1.1 OpenACC Overview

The parallelization of the project is based off of the nested for-loop in `renderFractal` that iterates over each pixel in the output image, calculating the correct colour for that pixel based off of the passed parameters. When the loop is encountered, the necessary data from the CPU's DRAM is copied in. A kernel is then launched on the GPU, and each thread begins the processing for one pixel. Every function call within the kernel is run sequentially on the calling thread, with all subsequent routines from *those* functions inlined. Finally, the image data is copied back to the CPU.

1.2 Vector3D

Originally, a `vec3` was represented by a C++ class. We changed this implementation to a struct, and wrote macros to perform computations on the vector. There were a few macros originally included with the project, but many more had to be written to ensure all `vec3` operations could be run on the `vec3` struct. Using macros simplified integration with OpenACC, as routines are a relatively new feature and not fully robust yet.

1.3 OpenACC Data

The complex data that is private to each thread, such as the pixel data objects, the vectors storing the `colour` and the `to` vector for the pixel, and the double array containing the `farPoint` for the pixel are all stored in arrays, where each thread on the GPU accesses one element of each array. These values are not needed by the CPU at all, so the arrays are allocated on GPU memory using `acc_malloc`, and declared as device pointers. Note that they are allocated once at the beginning of the program and freed just before the program exits.

1.4 Writing BMP to Disk

We observed that once the program renders an image, it must wait for the CPU to write that image to disk before continuing to the next frame. We introduced a simple optimization to allow execution to continue, so that the GPU can be rendering the next frame while the CPU is writing the previous one to the disk. This was a simple matter of creating a new thread to write the image out, then continuing to the next iteration for rendering, swapping out the image buffer with another. Once execution reaches the `saveBMP` call again, it waits for the "write-out" thread to finish, then spawns a new thread and continues. Having two image buffers does increase memory usage, but it allows both the host and device to do time-consuming work at the same time.

2 Computing Parameters for Frames

Generating the camera parameters for the next frame is done automatically, based on the furthest point in the mandelbox. To do this, additional data eneds to be stored about each point. That is, each GPU thread will know how far the point is that it hits from `rayMarch`, as well

as the vector representing that point. After the kernel has finished executing, we need to find the point witht the maximum distance (that is non-infinite). Initially, it seems intuitive to use an OpenACC `max` reduction to find this distance, but we need to know the vecotr associated with that point, which is not supproted by a reduction. Instead, we copy an array storing the distances to each point back from the GPU when the kernel exits, and the CPU iterates through them to find the max. Once it does, it uses `acc_memcpy_from_device` to copy the vector associated with that point back, without needing to copy the entire array of pixel data.

After the new vector is found, execution returns to `main`, and the CPU calculates camera position and target for the next frame. To make this smooth, every 10 frames the farthest point is saved, and for the other 9 iterations the frame's new target is linearly interpolated between the current point and the farthest point. The position of the camera always moves directly towards the target saved every 10 frames. This ensures that camera movement is smooth as the furthest points change, but the actual position of the camera will still go to the correct point (i.e. not hit a wall).

3   Mandelbox Performance vs. Mandelbulb

4   Source Code

The following source code files were not changed from the serial version of the program, and as such will their contents will not be reproduced in this document.

- `camera.h`
- `color.h`
- `getparams.c`
- `init3D.cc`
- `mandelbox.h`
- `renderer.h`
- `savebmp.c`

Due to OpenACC requirements for nested inline routines being in the same source file, the `MandelBoxDE` and `DE` functions were moved to `raymarching.cc`. Functions for printing progress and timing data were removed, as they were no longer deemed necessary due to the speedups from running the program on the GPU. As such, the following source code files were removed.

- `distance_est.cc`
- `mandelboxde.cc`
- `print.c`
- `timing.c`

Following are the modified source code files. Note that we made every effort to write "self-documenting" code by using clear variable names, so tried to limit comments within the source code to situations that are not immediately clear. Files are sorted alphabetically.

4.1 getcolor.cc

```cpp
/*
   This file is part of the Mandelbox program developed for the course
    CS/SE  Distributed Computer Systems taught by N. Nedialkov in the
    Winter of 2015-2016 at McMaster University.

    Copyright (C) 2015-2016 T. Gwosdz and N. Nedialkov

    This program is free software: you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation, either version 3 of the License, or
    (at your option) any later version.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public License
    along with this program.  If not, see <http://www.gnu.org/licenses/>.
*/

#include "color.h"
#include "renderer.h"
#include "vector3d.h"

//---lightning and colouring---------
#define CAM_LIGHT_W 1.8
#define CAM_LIGHT_MIN 0.3

#define CAM_LIGHT 1.0
#define BASE_COLOR 1.0
#define BACK_COLOR 0.4

inline void lighting(const vec3 &n, const vec3 &color, const vec3 &pos, const
    vec3 &direction,  vec3 &outV)
{
  vec3 nn;
  SUBTRACT_SCALAR(nn, n, 1.0);
  double d = DOT(direction, nn);
  double ambient = MAX(CAM_LIGHT_MIN, d) * CAM_LIGHT_W;
  vec3 camLight;
  VEC(camLight, CAM_LIGHT, CAM_LIGHT, CAM_LIGHT);
  MULT_SCALAR(nn, camLight, ambient);
  MULT_POINTWISE(outV, color, nn);
}

#pragma acc routine seq
inline void getColour(vec3 &hitColor, const pixelData &pixData, const
    RenderParams &render_params, const vec3 &from, const vec3 &direction)
```

```
{
  VEC(hitColor, BASE_COLOR, BASE_COLOR, BASE_COLOR);

  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_POINTWISE(hitColor, hitColor, pixData.normal);
      ADD_SCALAR(hitColor, hitColor, 1.0);
      DIV_SCALAR(hitColor, hitColor, 2.0);
      MULT_SCALAR(hitColor, hitColor, render_params.brightness);

      //gamma correction
      clamp(hitColor, 0.0, 1.0);
      MULT_POINTWISE(hitColor, hitColor, hitColor)
    }
    if (render_params.colourType == 1)
    {
     //"swap" colors
     double t = hitColor.x;
     hitColor.x = hitColor.z;
     hitColor.z = t;
    }
  }
  else
  {
    //we have the background colour
        VEC(hitColor, BACK_COLOR, BACK_COLOR, BACK_COLOR);
  }
}
```

4.2 main.cc

```
/*
   This file is part of the Mandelbox program developed for the course
   CS/SE  Distributed Computer Systems taught by N. Nedialkov in the
   Winter of 2015-2016 at McMaster University.

   Copyright (C) 2015-2016 T. Gwosdz and N. Nedialkov

   This program is free software: you can redistribute it and/or modify
   it under the terms of the GNU General Public License as published by
   the Free Software Foundation, either version 3 of the License, or
   (at your option) any later version.

   This program is distributed in the hope that it will be useful,
   but WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
   GNU General Public License for more details.

   You should have received a copy of the GNU General Public License
   along with this program.  If not, see <http://www.gnu.org/licenses/>.
*/
```

```c
#include <stdlib.h>

#include "camera.h"
#include "renderer.h"
#include "mandelbox.h"
#include "vector3d.h"
#include "color.h"
#include "openacc.h"

//#include <thread>
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>

void getParameters(char *filename, CameraParams *camera_params, RenderParams
    *renderer_params,
                   MandelBoxParams *mandelBox_paramsP);
void init3D       (CameraParams *camera_params, const RenderParams *
    renderer_params);
void renderFractal(const CameraParams camera_params, const RenderParams
    renderer_params, unsigned char* image);
void saveBMP      (const char* filename, const unsigned char* image, int
    width, int height);

#define NUM_FRAMES 10

void genNewCamParams(CameraParams &curCam, CameraParams &nextCam){
        curCam.camPos[0] += (nextCam.camPos[0] - curCam.camPos[0])*0.01;
        curCam.camPos[1] += (nextCam.camPos[1] - curCam.camPos[1])*0.01;
        curCam.camPos[2] += (nextCam.camPos[2] - curCam.camPos[2])*0.01;
}

vec3 newLookAt;

#pragma acc declare copyin(mandelBox_params)
MandelBoxParams mandelBox_params;

vec3* d_to;
vec3* d_colours;
double* d_farPoints;
pixelData* d_pixData;
double* d_distances;
#pragma acc declare deviceptr(d_to, d_colours, d_farPoints, d_pixData,
    d_distances)

int main(int argc, char** argv)
{
        // make directory to hold all the generated images
        struct stat st = {0};

        if (stat("/videoDir", &st) == -1) {
                mkdir("/videoDir", 0700);
        }
```

```cpp
  CameraParams    camera_params;
  RenderParams    renderer_params;

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

  int image_size = renderer_params.width * renderer_params.height;
  unsigned char *image1 = (unsigned char*)malloc(3*image_size*sizeof(unsigned
    char));
  unsigned char *image2 = (unsigned char*)malloc(3*image_size*sizeof(unsigned
    char));

  d_to = (vec3*)acc_malloc(image_size * sizeof(vec3));
  d_colours = (vec3*)acc_malloc(image_size * sizeof(vec3));
  d_farPoints = (double*)acc_malloc(image_size * 3 * sizeof(double));
  d_pixData = (pixelData*)acc_malloc(image_size * sizeof(pixelData));
  d_distances = (double*)acc_malloc(image_size * sizeof(double));

  vec3 newLookAtDest;
//  std::thread writeBMP;
  unsigned char *currImage;
  char new_file_name[80];

  for (int i = 0; i < NUM_FRAMES; i++) {
        if (i % 2 == 0) {
      currImage = image1;
    } else {
      currImage = image2;
    }

    init3D(&camera_params, &renderer_params);
    renderFractal(camera_params, renderer_params, currImage);
        printf("Done rendering frame %d... new lookAt = [%f,%f,%f]\n", i,
    newLookAt.x, newLookAt.y, newLookAt.z);

    if (i % 10 == 0) {
      newLookAtDest = newLookAt;
    }

    camera_params.camTarget[0] += (newLookAtDest.x - camera_params.camTarget
    [0])*0.1;
    camera_params.camTarget[1] += (newLookAtDest.y - camera_params.camTarget
    [1])*0.1;
    camera_params.camTarget[2] += (newLookAtDest.z - camera_params.camTarget
    [2])*0.1;

    // move towards point
    camera_params.camPos[0] += (newLookAtDest.x - camera_params.camPos[0])
    *0.01;
    camera_params.camPos[1] += (newLookAtDest.y - camera_params.camPos[1])
    *0.01;
    camera_params.camPos[2] += (newLookAtDest.z - camera_params.camPos[2])
    *0.01;

    if (i != 0) {
```

```
  //    writeBMP.join();
    }
    sprintf(new_file_name, "image_%d.bmp", i);
   // writeBMP = std::thread(saveBMP, new_file_name, currImage,
   renderer_params.width, renderer_params.height);
     saveBMP(new_file_name, currImage, renderer_params.width, renderer_params
    .height);
  }
  free(image1);
  free(image2);

  acc_free(d_to);
  acc_free(d_colours);
  acc_free(d_farPoints);
  acc_free(d_pixData);
  acc_free(d_distances);

  return 0;
}
```

4.3   raymarching.cc

```
/*
   This file is part of the Mandelbox program developed for the course
    CS/SE  Distributed Computer Systems taught by N. Nedialkov in the
    Winter of 2015-2016 at McMaster University.

    Copyright (C) 2015-2016 T. Gwosdz and N. Nedialkov

    This program is free software: you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation, either version 3 of the License, or
    (at your option) any later version.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public License
    along with this program.  If not, see <http://www.gnu.org/licenses/>.
*/

#include "color.h"
#include "renderer.h"
#include "mandelbox.h"
#include "vector3d.h"

#define SQR(x) ((x)*(x))
#define COMPONENT_FOLD(x) { (x) = (fabs(x) <= 1) ? (x) : (((x) > 0) ? (2-(x))
    : (-2-(x))); }

// #pragma acc declare copyin(mandelBox_params)
extern MandelBoxParams mandelBox_params;

extern double* d_distances;
```

```cpp
inline double MandelBoxDE(const vec3 &p0, const MandelBoxParams &params,
    double c1, double c2)
{
  vec3 p = p0;
  double rMin2   = SQR(params.rMin);
  double rFixed2 = SQR(params.rFixed);
  double escape  = SQR(params.escape_time);
  double dfactor = 1;
  double r2      = -1;
  const double 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_ASSIGN(r2,p);

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

    dfactor = dfactor * fabs(params.scale) + 1.0;
    MULT_SCALAR(p, p, params.scale);
    ADD_POINT(p, p, p0);
    i++;
  }

  return  (MAGNITUDE(p) - c1) / dfactor - c2;
}

inline double DE(const 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;
}

inline void normal(const vec3 & p, vec3 & normal)
{
  // compute the normal at p
  const double sqrt_mach_eps = 1.4901e-08;
  double eps = MAX( MAGNITUDE(p), 1.0 ) * sqrt_mach_eps;
```

```
    vec3 t1, e1;
    double x;

    VEC(e1, eps, 0, 0);
    ADD_POINT(t1, p, e1);
    x = DE(t1);
    SUBTRACT_POINT(t1, p, e1);
    normal.x = x - DE(t1);

    VEC(e1, 0, eps, 0);
    ADD_POINT(t1, p, e1);
    x = DE(t1);
    SUBTRACT_POINT(t1, p, e1);
    normal.y = x - DE(t1);

    VEC(e1, 0, 0, eps);
    ADD_POINT(t1, p, e1);
    x = DE(t1);
    SUBTRACT_POINT(t1, p, e1);
    normal.z = x - DE(t1);

    // calculating either of the last two x1,x2 causes compiler warning:
    // 'No device symbol for address reference'
    // This is dependent on order in source file only.

    NORMALIZE(normal);
}

#pragma acc declare copyin(mandelBox_params)
#pragma acc routine seq
void rayMarch(const RenderParams &render_params, const vec3 &from, const vec3
    &direction, double eps,
        pixelData& pix_data, double& distance)
{

    double dist = 0.0;
    double totalDist = 0.0;

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

    int steps = 0;
    vec3 p;
    do
    {
        MULT_SCALAR(p, direction, totalDist);
        ADD_POINT(p, p, from);
        dist = DE(p);

        totalDist += 0.95 * dist;

        epsModified = totalDist;
        epsModified *= eps;
        steps++;
    }
    while (dist > epsModified && totalDist <= render_params.maxDistance &&
```

```
  steps < render_params.maxRaySteps);

if (dist < epsModified)
{
  // we didnt escape
  pix_data.escaped = false;

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

  //figure out the normal of the surface at this point
  vec3 temp;
  MULT_SCALAR(temp, direction, epsModified);
  SUBTRACT_POINT(temp, p, temp);
  const vec3 normPos = temp;
  normal(normPos, pix_data.normal);
  distance = totalDist;
}
else {
  //we have the background colour
  pix_data.escaped = true;
  distance = 0.0;
}
}
```

4.4   renderer.cc

```
/*
   This file is part of the Mandelbox program developed for the course
   CS/SE  Distributed Computer Systems taught by N. Nedialkov in the
   Winter of 2015-2016 at McMaster University.

   Copyright (C) 2015-2016 T. Gwosdz and N. Nedialkov

   This program is free software: you can redistribute it and/or modify
   it under the terms of the GNU General Public License as published by
   the Free Software Foundation, either version 3 of the License, or
   (at your option) any later version.

   This program is distributed in the hope that it will be useful,
   but WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
   GNU General Public License for more details.

   You should have received a copy of the GNU General Public License
   along with this program.  If not, see <http://www.gnu.org/licenses/>.
*/
#include <stdio.h>
#include <stdlib.h>
#include "openacc.h"

#include "color.h"
#include "mandelbox.h"
#include "camera.h"
#include "vector3d.h"
#include "3d.h"
```

```cpp
extern double getTime();
extern void    printProgress( double perc, double time );

#pragma acc routine seq
extern void rayMarch (const RenderParams &render_params, const vec3 &from,
    const vec3  &to, double eps, pixelData &pix_data, double& distance);

#pragma acc routine seq
extern void getColour(vec3 & colour, const pixelData &pixData, const
    RenderParams &render_params, const vec3 &from, const vec3 &direction);

extern MandelBoxParams mandelBox_params;
extern vec3* d_to;
extern vec3* d_colours;
extern double* d_farPoints;
extern pixelData* d_pixData;

extern vec3 newLookAt;

void renderFractal(const CameraParams camera_params, const RenderParams
    renderer_params, unsigned char* image)
{
  vec3 fromTemp;
  VEC(fromTemp, camera_params.camPos[0], camera_params.camPos[1],
    camera_params.camPos[2]);
  const vec3 from = fromTemp;

  const double eps = pow(10.0, renderer_params.detail);

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

  double* distances = (double*)malloc(n * sizeof(double));

  #pragma acc data copyin(camera_params, renderer_params, eps, from)
  #pragma acc data deviceptr(d_to, d_colours, d_farPoints, d_pixData)
  #pragma acc data copyout(image[:n*3], distances[:n])
  {
    #pragma acc kernels loop independent collapse(2)
    for(int j = 0; j < height; j++)
    {
      for(int i = 0; i < width; i++)
      {
        int k = j*width + i;

        UnProject(i, j, camera_params, &(d_farPoints[k*3]));

        SUBTRACT_DARRS(d_to[k], (&(d_farPoints[k*3])), camera_params.camPos);
        NORMALIZE(d_to[k]);

        rayMarch(renderer_params, from, d_to[k], eps, d_pixData[k], distances
    [k]);
        getColour(d_colours[k], d_pixData[k], renderer_params, from, d_to[k])
    ;
```

13

```c
        //save colour into texture
        image[k*3 + 2] = (unsigned char)(d_colours[k].x * 255);
        image[k*3 + 1] = (unsigned char)(d_colours[k].y * 255);
        image[k*3]     = (unsigned char)(d_colours[k].z * 255);
      }
    }
  }

  // find the index of the farthest point
  double maxDistance = 0;
  int maxDistanceIndex = -1;
  for (int i = 0; i < n; i++) {
    if (distances[i] > maxDistance) {
      maxDistance = distances[i];
      maxDistanceIndex = i;
    }
  }

  // copy the vector at that point to our new look at
  if (maxDistanceIndex >= 0) {
      acc_memcpy_from_device(&newLookAt, &(d_pixData[maxDistanceIndex].hit),
    sizeof(vec3));
  } else {
    printf("No distance greater than 0 found. Looking at [0,0,0].\n");
    newLookAt.x = 0;
    newLookAt.y = 0;
    newLookAt.z = 0;
  }

  free(distances);
}
```

4.5   vector3d.h

```c
#ifndef vec3_h
#define vec3_h

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

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

#define SET_POINT(p,v) { p.x=v[0]; p.y=v[1]; p.z=v[2]; }

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

```c
#define SUBTRACT_DARRS(p,d1,d2) {\
  p.x=(d1[0])-(d2[0]);\
  p.y=(d1[1])-(d2[1]);\
  p.z=(d1[2])-(d2[2]);\
}


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


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


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


#define MAGNITUDE(p) ({ sqrt( p.x*p.x + p.y*p.y + p.z*p.z ); })

#define DOT_ASSIGN(d,p) {  d=( p.x*p.x + p.y*p.y + p.z*p.z ); }
#define DOT(p,q) { (p.x*q.x + p.y*q.y + p.z*q.z) }
#define MAX(a,b) ( ((a)>(b)) ? (a) : (b))

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

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


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


#define DIV_SCALAR(p,v,s) {\
        double fInv = 1.0 / s;\
        p.x = v.x * fInv;\
        p.y = v.y * fInv;\
        p.z = v.z * fInv;\
```

```
}

inline double clamp(double d, double min, double max)
{
  const double t = d < min ? min : d;
  return t > max ? max : t;
}

inline void clamp(vec3 &v, double min, double max)
{
  v.x = clamp(v.x,min,max);
  v.y = clamp(v.y,min,max);
  v.z = clamp(v.z,min,max);
}

#endif
```

4.6   3d.cc

```
/*
   This file is part of the Mandelbox program developed for the course
    CS/SE  Distributed Computer Systems taught by N. Nedialkov in the
    Winter of 2015-2016 at McMaster University.

    Copyright (C) 2015-2016 T. Gwosdz and N. Nedialkov

    This program is free software: you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation, either version 3 of the License, or
    (at your option) any later version.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public License
    along with this program.  If not, see <http://www.gnu.org/licenses/>.
*/
// #include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define PI 3.14159265358979323846 // M_PI not defined in accelmath
#ifdef _OPENACC
  #include <accelmath.h>
#else
  #include <math.h>
#endif

#include "3d.h"

//
    ---------------------------------------------------------------------------------

//when projection and modelview matricies are static (computed only once, and
```

```
     camera does not mover)
int UnProject(double winX, double winY, const CameraParams& camP, double *obj
    )
{
  //Transformation vectors
  double in[4], out[4];

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

  //Objects coordinates
  MultiplyMatrixByVector(out, camP.matInvProjModel, 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(double *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(double fov, double aspect, double zNear, double zFar, double
    *projMat)
{
  double ymax, xmax;
```

```
  ymax = zNear * tan(fov * PI / 360.0);
  //ymin = -ymax;
  //xmin = -ymax * aspectRatio;
  xmax = ymax * aspect;
  Frustum(-xmax, xmax, -ymax, ymax, zNear, zFar, projMat);
}


void Frustum(double left, double right, double bottom, double top, double
    znear, double zfar, double *matrix)
{
  double 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(double *eye, double *target, double *upV, double *modelMatrix)
{
  double forward[3], side[3], up[3];
  double 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;
  //-----------------
```

```c
  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(double));
}


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

void ComputeNormalOfPlane(double *normal, double *v1, double *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(double *result, const double *matrix1, const double *
    matrix2)
{
  result[0]=matrix1[0]*matrix2[0]+
    matrix1[4]*matrix2[1]+
    matrix1[8]*matrix2[2]+
    matrix1[12]*matrix2[3];
  result[4]=matrix1[0]*matrix2[4]+
    matrix1[4]*matrix2[5]+
    matrix1[8]*matrix2[6]+
    matrix1[12]*matrix2[7];
  result[8]=matrix1[0]*matrix2[8]+
    matrix1[4]*matrix2[9]+
    matrix1[8]*matrix2[10]+
    matrix1[12]*matrix2[11];
  result[12]=matrix1[0]*matrix2[12]+
    matrix1[4]*matrix2[13]+
    matrix1[8]*matrix2[14]+
    matrix1[12]*matrix2[15];
  result[1]=matrix1[1]*matrix2[0]+
    matrix1[5]*matrix2[1]+
```

```
      matrix1[9]*matrix2[2]+
      matrix1[13]*matrix2[3];
    result[5]=matrix1[1]*matrix2[4]+
      matrix1[5]*matrix2[5]+
      matrix1[9]*matrix2[6]+
      matrix1[13]*matrix2[7];
    result[9]=matrix1[1]*matrix2[8]+
      matrix1[5]*matrix2[9]+
      matrix1[9]*matrix2[10]+
      matrix1[13]*matrix2[11];
    result[13]=matrix1[1]*matrix2[12]+
      matrix1[5]*matrix2[13]+
      matrix1[9]*matrix2[14]+
      matrix1[13]*matrix2[15];
    result[2]=matrix1[2]*matrix2[0]+
      matrix1[6]*matrix2[1]+
      matrix1[10]*matrix2[2]+
      matrix1[14]*matrix2[3];
    result[6]=matrix1[2]*matrix2[4]+
      matrix1[6]*matrix2[5]+
      matrix1[10]*matrix2[6]+
      matrix1[14]*matrix2[7];
    result[10]=matrix1[2]*matrix2[8]+
      matrix1[6]*matrix2[9]+
      matrix1[10]*matrix2[10]+
      matrix1[14]*matrix2[11];
    result[14]=matrix1[2]*matrix2[12]+
      matrix1[6]*matrix2[13]+
      matrix1[10]*matrix2[14]+
      matrix1[14]*matrix2[15];
    result[3]=matrix1[3]*matrix2[0]+
      matrix1[7]*matrix2[1]+
      matrix1[11]*matrix2[2]+
      matrix1[15]*matrix2[3];
    result[7]=matrix1[3]*matrix2[4]+
      matrix1[7]*matrix2[5]+
      matrix1[11]*matrix2[6]+
      matrix1[15]*matrix2[7];
    result[11]=matrix1[3]*matrix2[8]+
      matrix1[7]*matrix2[9]+
      matrix1[11]*matrix2[10]+
      matrix1[15]*matrix2[11];
    result[15]=matrix1[3]*matrix2[12]+
      matrix1[7]*matrix2[13]+
      matrix1[11]*matrix2[14]+
      matrix1[15]*matrix2[15];
}

inline void MultiplyMatrixByVector(double *resultvector, const double *matrix
    , double *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) { double *_tmp = a; (a)=(b); (b)=_tmp; }
#define MAT(m,r,c) (m)[(c)*4+(r)]

int InvertMatrix(double *m, double *out){
  double wtmp[4][8];
  double m0, m1, m2, m3, s;
  double *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 (fabs(r3[0]) > fabs(r2[0]))
    SWAP_ROWS(r3, r2);
  if (fabs(r2[0]) > fabs(r1[0]))
    SWAP_ROWS(r2, r1);
  if (fabs(r1[0]) > fabs(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;
  }
```

```c
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 (fabs(r3[1]) > fabs(r2[1]))
  SWAP_ROWS(r3, r2);
if (fabs(r2[1]) > fabs(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 (fabs(r3[2]) > fabs(r2[2]))
  SWAP_ROWS(r3, r2);
if (0.0 == r2[2])
```

```c
    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(double *result, double x, double y, double z){
  double matrix[16], resultMatrix[16];

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

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

```
}
```

4.7 3d.h

```
/*
   This file is part of the Mandelbox program developed for the course
    CS/SE  Distributed Computer Systems taught by N. Nedialkov in the
    Winter of 2015-2016 at McMaster University.

    Copyright (C) 2015-2016 T. Gwosdz and N. Nedialkov

    This program is free software: you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation, either version 3 of the License, or
    (at your option) any later version.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public License
    along with this program.  If not, see <http://www.gnu.org/licenses/>.
*/

#ifndef _3d_H
#define _3d_H

#define NEAR 1
#define FAR 100

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


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

#pragma acc routine seq
int    UnProject(double winX, double winY, const CameraParams & camP, double
```

```
    *obj);
```

```
#endif
```

5   Running the Program

To generate the video, follow the following steps.

1. Run `make mandelbox` from the project directory

2. Execute the program with `./mandelbox params.dat`

3. Convert the images to a video by running `convert_to_video`

If you wish to change the number of frames, simply open `main.cc` and replace the value for the `NUM_FRAMES` pragma with the desired number of frames.

6   Bonus Features

6.1   Automatic Navigation

The automatic navigation functionality for the project was implemented. That is, the pgroam will determine a path through the mandelbox that does not "hit" any walls, nor leave the box itself. For more details on the algorithm used, refer to section 2.