Computer Science

COMPSCI 373 S1 C 2015

Making a Ray Caster: Part 1

The work done on this task must be your own work. Think carefully about any problems you come across, and try to solve them yourself before you ask anyone else for help. Under no circumstances should you work together with another student to solve problems posed in this task.

Assignment

This week's assignment will be composed of two Quizzes in coderunner. The theoretical Quiz will have 5 questions each worth 0.25 marks. You will have 1 hour (and 3 repeats) to answer it. The programming Quiz will have 5 small programming/practical questions each worth 0.25 marks. You will have 2 hours (and 3 repeats) to answer it. Both theoretical and practical questions will query about the implementation of a Ray Casting engine based on the 2D Ray Casting section of your lectures. This document proposes a step-by-step implementation of a ray caster in line with the supporting material provided. It should help you prepare for the programming Quiz and better understand the principles of a Ray Caster. It is up to you to complete the below tasks. There is no grade attached to the completion of the skeleton code provided here.

Included resources

You will be provided with the skeleton code for the Ray Casting engine. Initially, it will draw an overhead map of the world. As you complete more of the skeleton code, more of the engine will become functional.

You are provided with the following files:

main.c…….The main program entry point and routines
373assignment.h……The Ray Caster program parameters and environment definitions.
linalg.c/linalg.h ……The linear algebra (vector/matrix) operations library
map.c/map.h …….The overhead map drawing code
player.c/player.h……The player control code
raycaster.c/raycaster.h…..The Ray Casting stepping algorithm functions
renderer.c/renderer.h……The screen rendering functions

In order to fully complete the skeleton code and have a fully functional Ray Casting engine, you will need to make changes to the following files:

linalg.c
player.c
raycaster.c
renderer.c

Stage 1: The linear algebra functions

The linear algebra library provides the mathematical basis for the rest of the Ray Casting engine. The library defines two types, and 10 functions. The two types are:

```
typedef struct {
    float x;
    float y;
    float z;
} Vector3f;
```

and

```
typedef float Matrix3f[3][3]
```

Vector3f is a 2D homogeneous vector (which is why it has three components), and Matrix3f, which is a homogeneous 2D matrix (which is why it has 3x3 elements).
The library defines 10 functions which operate with these two types:

```
Vector3f homogeneousVectorAdd(Vector3f vec1, Vector3f vec2);
Vector3f homogeneousVectorSubtract(Vector3f vec1, Vector3f vec2);
Vector3f homogeneousVectorScale(Vector3f vec, float scalar);
float homogeneousVectorMagnitude(Vector3f vec);
Vector3f normalizeHomogeneousVector(Vector3f vec);
float homogeneousVectorDotProduct(Vector3f vec1, Vector3f vec2);
Vector3f homogeneousVectorProjection(Vector3f vec1, Vector3f vec2);
Vector3f homogeneousReciprocalVectorProjection(Vector3f vec1, Vector3f vec2);
void matrixVectorMultiply(Matrix3f* mat, Vector3f* vec);
void    matrix3fCopy(Matrix3f* dst, Matrix3f* src);
```

homogeneousVectorAdd, homogeneousVectorMagnitude, and matrix3fCopy are provided for you. The details of each function can be found in *linalg.c*, as comments above each respective function.

Once you have completed the remaining 7 functions, you can check their correctness by copying your implementations into the skeleton linear algebra project provided for you here:

https://www.cs.auckland.ac.nz/courses/compsci373s1c/tutorials/Tutorials2015/LinearAlgebra.zip

The project includes test cases for every function.It is important that you check your code here to make sure that your implementations are correct. Once they are, you can move on to *Stage 2*.

Stage 2: Initializing the Ray Caster

Your next step is to complete the Ray Caster initialization function, **setupRaycaster()**. This function can be found inside *raycaster.c*.

The purpose of this function is to set the distance to the viewplane, as well as initialize the two player rotation matrices (one for clockwise, and one for counter-clockwise rotations). The distance of the viewplane must be derived using the provided initial Field-Of-View value, which is given in the global variable FOV as an angle in radians. Details on how to do this are provided in your lecture notes.

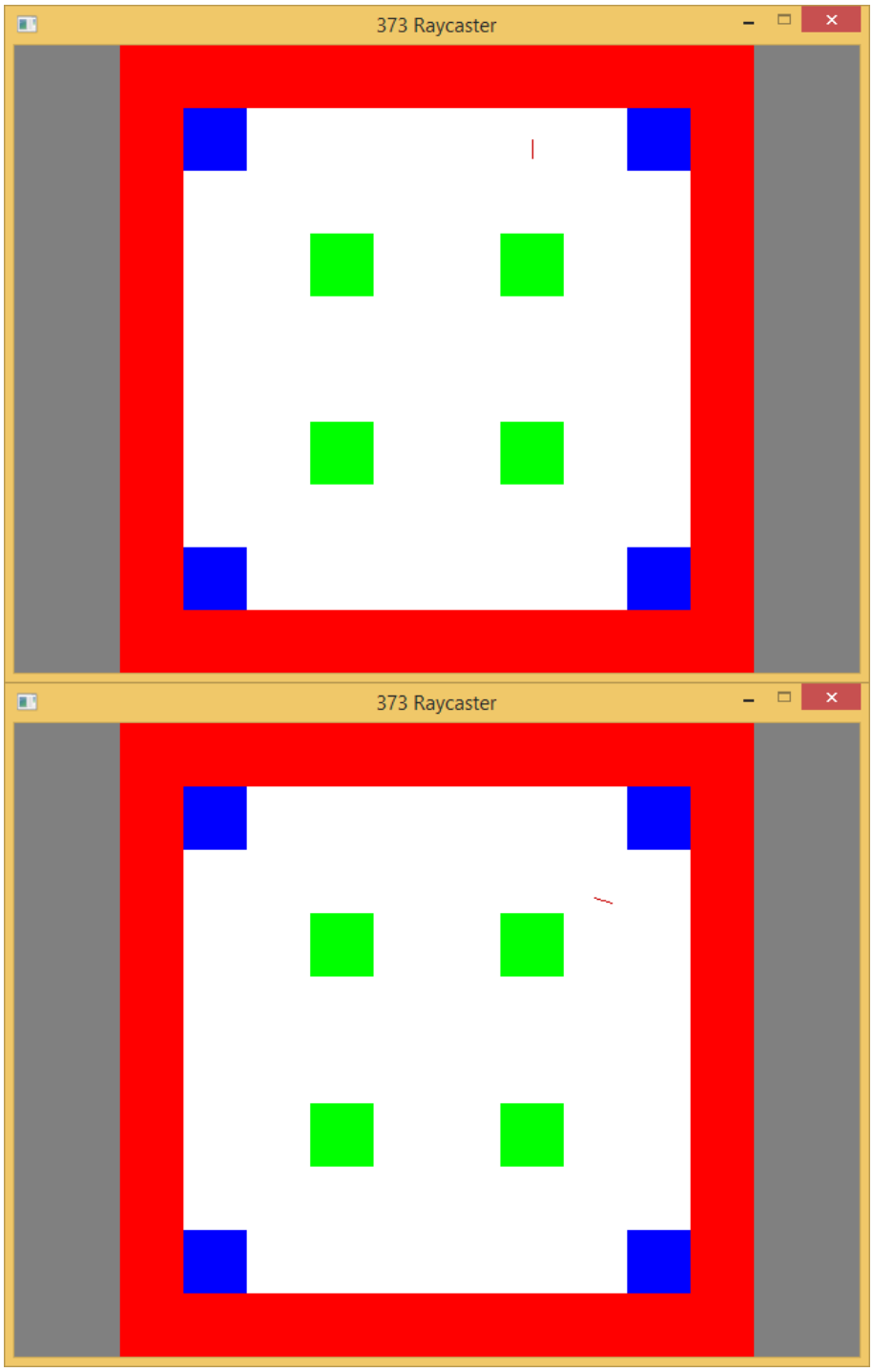
When calculating the rotation matrices, you must use the theta value provided, which is given in a global variable named *PLAYER_ROT_SPEED*.

Once you have completed this function, you can move on to *Stage 3*.

Stage 3: Player rotation

Your goal in this stage is to provide the function to rotate the projection environment by a given rotation matrix (there is more than one vector you will need to rotate). You will need to make use of the linear algebra library, as well.
For a hint, recall the vectors that define the projection environment. How would a rotation affect these?
You will need to apply a rotation to a few of the projection environment vectors. The names of all the provided vectors and parameters are in the appendix of this handout.

Once you complete this stage, you should be able to move the player around the map. Here is an example of what you would see before and after pressing the forward and left arrow keys for a short time:



Stage 4: Initializing rays

Now you are ready to start implementing the Ray Casting functions. The first function you will need to implement is the one which initializes all of the ray vectors for a frame so that they are all distributed across the entire viewplane. To do this, you should make use of the formula given to you in your lecture notes.

To implement the formula, you will need to compute the values of at least three different vectors, these will be:

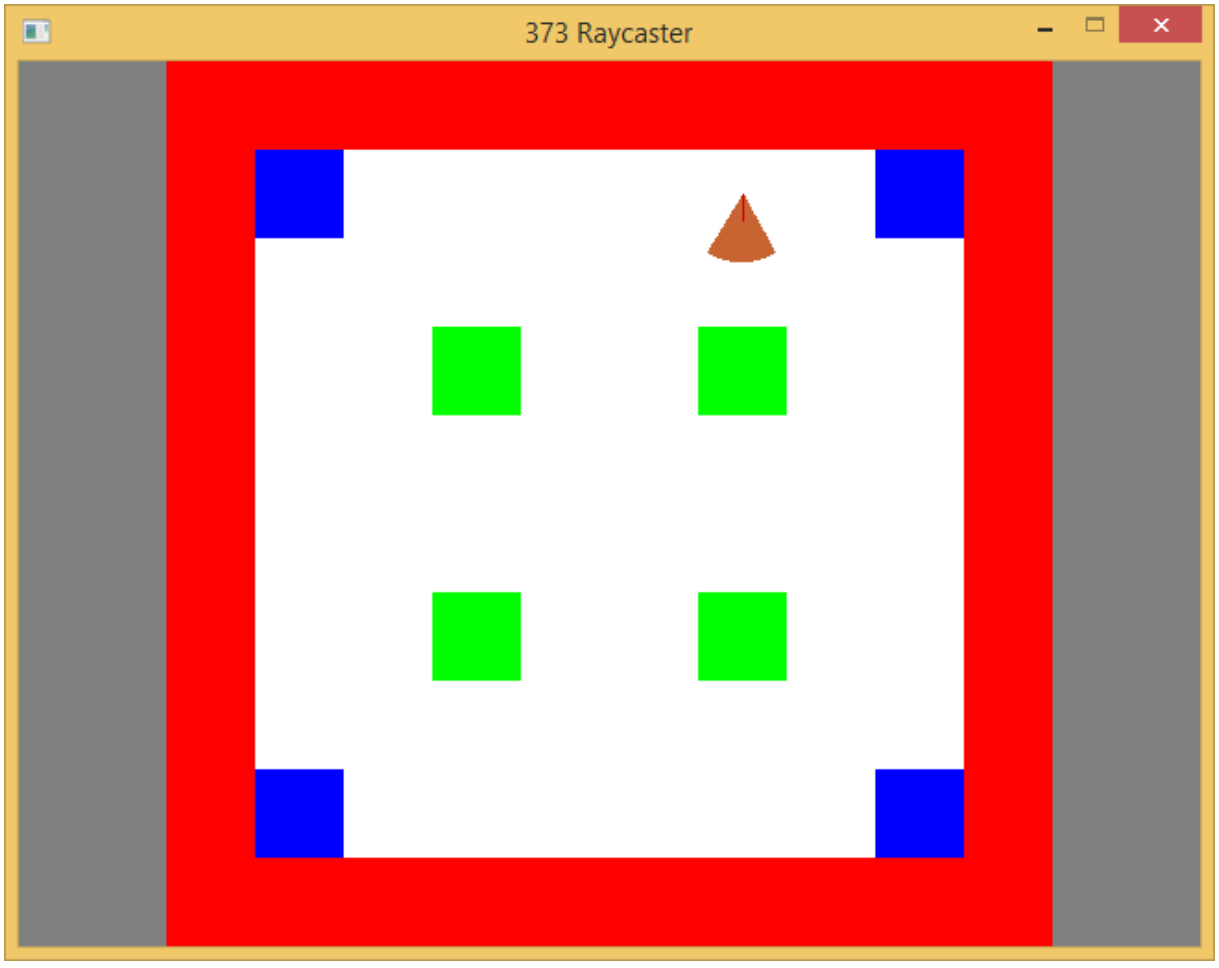
Vector3f vdd - The vector pointing from the camera to the viewplane, with the magnitude being the distance of the camera from the viewplane.

Vector3f vn - The vector required to translate vdd to the point on the viewplane where the nth ray needs to point.

Vector3f rn - The nth ray vector.

Once you have **rn**, you must make it viewable by the rest of the program. This is done by storing it in the nth element of the **rays[]** array.

If you run your program at this point, the rays will be drawn, but because they should be very small, you shouldn't be able to see them yet. In order to debug your program to make sure that it works up to this point, you can use the *homogeneousVectorScale* function to scale each vector by a factor. If you scale the vectors by a factor of 50, you should be able to see something like the following:



Make sure that the rays rotate correctly with the player. Once you're sure everything works, **remember to remove the extra scaling function.**

You should now be able to move on to *Stage 5*.

Stage 5: Calculating the stepping vectors

Your task in this stage is to implement the functions which, when given a ray, calculate the initial stepping vectors, $\boldsymbol{H}_i$ and $\boldsymbol{V}_i$, as well as the main stepping vectors $\boldsymbol{H}_s$ and $\boldsymbol{V}_s$.

The functions you need to implement are:

```
Vector3f findInitialVerticalRayStepVector(Vector3f rayVec, Vector3f rayStart);
Vector3f findInitialHorizontalRayStepVector(Vector3f rayVec, Vector3f rayStart);
Vector3f findVerticalRayStepVector(Vector3f rayVec);
Vector3f findHorizontalRayStepVector(Vector3f rayVec);
```

In the case of the first two functions, *rayStart* is always going to be a vector representing the x,y coordinate of the camera within the world. In the case of all functions, *rayVec* is the vector representing the normalized ray that we want to find stepping vectors for.

Your task for the first two functions is to first find the $\boldsymbol{S}_V$ and $\boldsymbol{S}_H$ vectors. The details of these two vectors is given in your lecture slides. In order to find these, you will need to find the vector between the player's position and one of the nearest walls. In order to assist you, we provide code for calculating the x coordinate of the nearest wall in the negative x direction (*westX*), and the y coordinate of the nearest wall in the negative y direction (*northY*).

For the functions that calculate the main stepping vectors, $\boldsymbol{S}_V$ and $\boldsymbol{S}_H$ can be calculated using only the width of a grid square, which is given in the variable *WALL_SIZE*.

Once you have completed all four functions, you can move on to *Stage 6*.

Stage 6: Performing Ray Casting

Your task for this stage is to complete the implementation of the raycasting function:

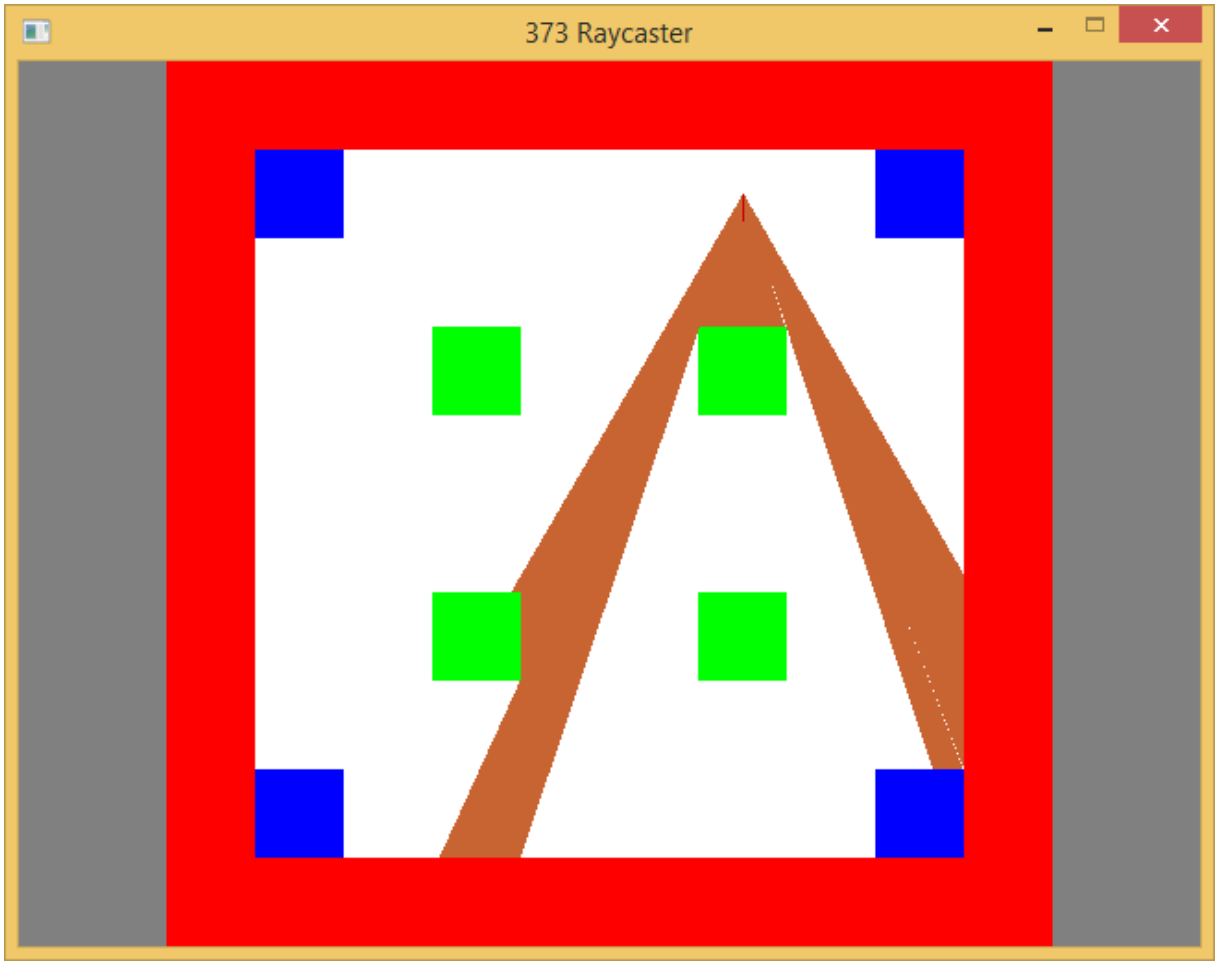
```
Ray raycast(Ray ray, Vector3f rayStart);
```

This function performs the entire ray stepping algorithm for vertical and horizontal hits along a ray, and returns the vector to the nearest intersection.

The first step in implementing this function is to define the initial horizontal and vertical step vectors. You can do this by making use of the functions that you implemented for *Stage 5*. Next, you will need to define the initial values for the *vray* and *hray* vectors. These correspond to the $\boldsymbol{R}_{vn}$ and $\boldsymbol{R}_{hn}$ vectors in your slides respectively. They will need to be initially set to their first intersection locations, which will be either $\boldsymbol{H}_i$ or $\boldsymbol{V}_i$. Afterwards, you need to step along the vectors using the *vstep* and *hstep* vectors until an intersection is reached.

Once both types of intersections are reached, you need to compare the magnitude of the two rays, and set *ray.vec* to be the smaller of the two.

If this stage is working correctly, you should see the following:



Once you're satisfied with your implementation, congratulations, you have completed this week's portion of the skeleton code!

Appendix

There are a number of global variables which you must use when implementing the assignment. A global variable is one that is accessible everywhere in the program, even in different .c files. Below is a list of important global variables, organized into categories:

PROJECTION ENVIRONMENT

Vector3f viewplaneDir…...The normalized viewplane direction vector
float distFromViewplane…The distance of the camera to the viewplane
Vector3f playerDir……..The normalized player (camera) direction vector
Vector3f playerPos……..The x,y position of the player (camera) in the world
Ray rays[]……..……..The array of rays for the current projection environment

MISC.

Matrix3f counterClockwiseRotation…The counterclockwise player rotation matrix
Matrix3f clockwiseRotation……The clockwise player rotation matrix

The file *373assignment.h* also provides some important constants:

```
/* Window parameters*/
WINDOW_WIDTH  640  /* The width of the rendering viewport */
WINDOW_HEIGHT 480  /* The height of the rendering viewport */

/* Raycaster parameters */
TEXTURE_SIZE 64  /* The size of a square wall texture in pixels */
WALL_SIZE 64  /* The size of a wall cube face in the world in pixels */
FOV (PI / 3.0f) /* 60 degrees */
PLAYER_MOVEMENT_SPEED 5.0f /* 5 pixels per frame */
PLAYER_ROT_SPEED ((3.0f * (PI)) / 180.0f) /* 3 degrees per frame */
PLAYER_SIZE 20 /* Player is 20 pixels wide*/

/* Projection parameters */
VIEWPLANE_LENGTH  WINDOW_WIDTH/*The viewplane length is the same as the window width*/
VIEWPLANE_DIR {-1, 0, 1} /*The initial viewplane direction vector*/
PLAYER_DIR {0, 1, 1} /* The initial player direction vector */
PLAYER_START {(2.5f * WALL_SIZE), (2.5f * WALL_SIZE), 1} /* Default player start
coordinate */
```

Note that these care constant (cannot be changed), and many of them (such as PLAYER_DIR) are only used to initialize non constant variables, and shouldn't be used for anything else.